Report Number: WUCSE-2005-18

2005-05-01

Architectures for Rule Processing Intrusion Detection and Prevention Systems

Michael E. Attig

High-performance intrusion detection and prevention systems are needed by network administrators in order to protect Internet systems from attack. Researchers have been working to implement components of intrusion detection and prevention systems for 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 intrusion detection and prevention for real network systems. In this thesis, three architectures to perform rule processing, the heart of intrusion detection and prevention, are presented.... **Read complete abstract on page 2.**

Architectures for Rule Processing Intrusion Detection and Prevention Systems

Michael E. Attig

Complete Abstract:

High-performance intrusion detection and prevention systems are needed by network administrators in order to protect Internet systems from attack. Researchers have been working to implement components of intrusion detection and prevention systems for 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 intrusion detection and prevention for real network systems. In this thesis, three architectures to perform rule processing, the heart of intrusion detection and prevention, are presented. The first system, called Snort Lite, implements a subset of the features necessary for rule processing in a single Xilinx Virtex XCV2000E _eld programmable gate array. The second system, called Snort Intrusion Filter for TCP (SIFT), limits the amount of traffic an intrusion detection PC needs to examine by searching for rule criteria. The final architecture presents a framework for implementing the entire rule processing system in reconfigurable hardware. The framework integrates the functionality to scan data flows for regular expressions, _xed strings, and header values. Additional processing modules can be added to the system to perform specific functionality required for some Snort rules. Reconfigurability and flexibility are key features of the system that enable it to adapt to protect Internet systems from threats including malicious worms, computer viruses, and network intruders. The framework allows experimentation with new techniques to perform the functionality required for intrusion systems. Each architecture uses the Field-programmable Port eXtender (FPX) platform to scan all bytes of Transmission Control Protocol/Internet Protocol (TCP/IP) traffic entering and leaving a network's gateway at multi-gigabit rates. The combined circuits perform deep-packet inspection to search for thousands of signatures. The rule processing framework supports up to 32,768 complex rules at data rates of 2.5 Gbps on the FPX platform.

2005-18

Architectures for Rule Processing Intrusion Detection and Prevention Systems, Master's Thesis, May 2005

Authors: Attig, Michael E.

April 29, 2005

Abstract: High-performance intrusion detection and prevention systems are needed by network administrators in order to protect Internet systems from attack. Researchers have been working to implement components of intrusion detection and prevention systems for 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 intrusion detection and prevention for real network systems.

In this thesis, three architectures to perform rule processing, the heart of intrusion detection and prevention, are presented. The first system, called Snort Lite, implements a subset of the features necessary for rule processing in a single Xilinx Virtex XCV2000E field programmable gate array. The second system, called

SEVER INSTITUTE OF TECHNOLOGY

MASTER OF SCIENCE DEGREE

THESIS ACCEPTANCE

(To be the first page of each copy of the thesis)

DATE: April 28, 2005

STUDENT'S NAME: Michael E. Attig

This student's thesis, entitled Architectures for Rule Processing Intrusion Detection and Prevention Systems has been examined by the undersigned committee of three faculty members and has received full approval for acceptance in partial fulfillment of the requirements for the degree Master of Science.

APPROVAL: _____ Chairman

_____

_____

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ARCHITECTURES FOR RULE PROCESSING INTRUSION DETECTION AND PREVENTION SYSTEMS

by

Michael E. Attig

B.S. Computer Engineering, B.S. Electrical Engineering

Prepared under the direction of Professor John W. Lockwood

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

May, 2005

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

ARCHITECTURES FOR RULE PROCESSING INTRUSION DETECTION AND PREVENTION SYSTEMS

by Michael E. Attig

ADVISOR: Professor John W. Lockwood

May, 2005
Saint Louis, Missouri

High-performance intrusion detection and prevention systems are needed by network administrators in order to protect Internet systems from attack. Researchers have been working to implement components of intrusion detection and prevention systems for 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 intrusion detection and prevention for real network systems.

In this thesis, three architectures to perform rule processing, the heart of intrusion detection and prevention, are presented. The first system, called Snort Lite,

implements a subset of the features necessary for rule processing in a single Xilinx Virtex XCV2000E field programmable gate array. The second system, called Snort Intrusion Filter for TCP (SIFT), limits the amount of traffic an intrusion detection PC needs to examine by searching for rule criteria. The final architecture presents a framework for implementing the entire rule processing system in reconfigurable hardware. The framework integrates the functionality to scan data flows for regular expressions, fixed strings, and header values. Additional processing modules can be added to the system to perform specific functionality required for some Snort rules. Reconfigurability and flexibility are key features of the system that enable it to adapt to protect Internet systems from threats including malicious worms, computer viruses, and network intruders. The framework allows experimentation with new techniques to perform the functionality required for intrusion systems.

Each architecture uses the Field-programmable Port eXtender (FPX) platform to scan all bytes of Transmission Control Protocol/Internet Protocol (TCP/IP) traffic entering and leaving a network's gateway at multi-gigabit rates. The combined circuits perform deep-packet inspection to search for thousands of signatures. The rule processing framework supports up to 32,768 complex rules at data rates of 2.5 Gbps on the FPX platform.

To my family and friends,
who have always supported me.

Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my advisor, Dr. Lockwood, for guiding me through the graduate school process. First, he supported my research. Second, his comments and suggestions helped me to fully develop ideas.

Additionally, I would like to thank the research sponsors Global Velocity and Boeing for not only the financial backing but also for the experience working with them has given me.

This work could not have been completed without the help, support, and patience of the members of the Reconfigurable Network Group in the Applied Research Laboratory.

Thanks are due to Todd Sproull for his numerous explanations about how to configure the lab; Sarang Dharmapurikar for explaining how to use a Bloom filter; Dave Lim for his work on the NID and tireless efforts to show me the problem is in my circuit; Dave Schuehler for tools to collect statistics and generate graphs as well as for the TCP wrapper; Haoyu Song for his help in configuring the Gigabit Ethernet line card; James Moscola for explaining how to use the IP wrappers; Phillip Jones for his numerous discussions and putting up with my ramblings; and finally to the remainder of the graduate students in the group (past and present) whose discussions and suggestions have been very helpful: Adam Covington, Chip Kastner, Andrew Levine, Jing Lu, Bharath Madhusudan, Jack Meier, Jeff Mitchell, Chris Neely, Shobana Padmanabhan, Abdel Rigumye, Dave Taylor, Qian Wan, and Chris Zuver.

Michael E. Attig

Washington University in Saint Louis
May 2005

Chapter 1

Introduction

The world is now networked together. People, businesses, and governments share information and communicate nearly instantaneously. Individuals use the network for today's everyday tasks, such as banking, shopping, investing, or transferring pictures to friends. With sensitive information now available on-line, measures must be taken to ensure security and privacy. The electronic database of customer's credit card numbers, addresses, and phone numbers must be secured against identity theft. Similarly, medical institutions must secure patient information to protect medical information and maintain privacy.

High-speed network connections have dramatically increased the amount of information that can be communicated. However, these same high speed links have also aided the spread of malicious software. This 'malware' has been used to cripple businesses and bring servers to a standstill. Worm attacks by Nimda, Code Red, Slammer, SoBigF, and MSBlast have infected computers globally, clogged large computer networks, and degraded productivity, costing *billions* of dollars.

In order to protect networked systems, intrusion detection and prevention is necessary. Intrusion detection determines when harmful activities are being attempted. Intrusion prevention systems stop malware from infecting additional machines over the network. In the remainder of this chapter, the need for high speed rule processing systems is motivated, intrusion detection is defined, and the notion of rule processing for intrusion detection is explored. Finally, the contributions of the work are presented.

1.1 Motivation

Intrusions are unlikely to stop anytime soon. As more people, institutions, governments, and companies are networked together, the threat of intrusions increases. It is exceedingly difficult to build and maintain systems that are totally foolproof and have no security holes. Due to time-to-market constraints and because it takes too long to test all possible permutations of events that can produce an intrusion, it is unlikely that systems will ever be built that are totally secure. Network systems that can detect intrusions and prevent future intrusions are critical for security.

The work that was performed for this thesis had two main motivating factors. First, the monetary costs of intrusions, such as from spam, worms, and viruses is exponentially increasing. Solutions are needed now to combat this growing trend. Second, current rule processing systems are almost entirely implemented using software that runs on processors that cannot scale to process data on fast links. Hardware implementations allow for higher throughput, increase rule capacity, and take advantage of the parallelism that is inherent in rule processing.

1.1.1 Economic Impact of Intrusions and Spam

The proliferation of spam and malicious software circulating on the public network is extraordinary. First, the lost productivity costs institutions billions of dollars per year. Spam messages (that get by filters) require time to be deleted by the human recipients. The emergence of the billion-dollar security and spam industry is proof that this is a wide-scale problem, as evidenced by the emergence of companies like *Symantec* and *McAfee* and projects like *SpamAssassin*.

Second, the severity of malware forces companies to devote entire departments of full-time staff members to ensure that computing systems are patched with the latest software. Consider the MSBlast worm, a particularly annoying worm that affected more than eight million vulnerable Windows-based machines [63]. This worm rebooted computers after a minute of being operational, which was too little time to recognize the problem and install the patch. Future worms are expected to be more dangerous and could modify or delete data on infected computers. The estimated costs of certain worm and virus instances are summarized in Table 1.1. Nearly all major worm or virus outbreaks have cost billions of dollars [1, 123].

Table 1.1: A summary of the cost of malicious code by incident [1, 123].

Year	Incident	Worldwide Impact (US Dollars)
2004	Netsky	$11 Billion
	Sasser	$6.25 Billion
	Mydoom.F	$22.6 Billion
2003	Sobig.F	$2 Billion
	Blaster	$1.3 Billion
	Slammer	$1.2 Billion
2002	KLEZ	$9 Billion
2001	Nimda	$635 Million
	Code Red	$2.62 Billion
	SirCam	$1.15 Billion
2000	Love Bug	$8.75 Billion
1999	Melissa	$1.1 Billion
	Explorer	$1.02 Billion

Finally, recent scams seek information from recipients. This tactic, known as phishing, has become very sophisticated and is difficult to prevent. In such cases, people unknowingly give sensitive information directly to the phishers who fraudulently pose as a reputable institution such as *PayPal*, *Citibank*, or *Washington Mutual*. In some cases, entire bank accounts are emptied and people are left wondering what happened. This scheme has become a major issue for law enforcement agencies. Many of the sources of such malware and phishing scams originate in countries with few laws to prevent such actions. The estimated worldwide impact of malware has grown exponentially, as shown in Figure 1.1.

1.1.2 Scalability

Databases of rules have been developed to identify malware. As new threats emerge, these rules increase in complexity. It was shown in [94] that the current volume of rules cannot be processed completely in software using today's PCs even at 100 Mbps rates. As link speeds increase to 1 Gbps, OC-48 (2.5 Gbps), OC-192 (10 Gbps), and beyond, systems are needed that accelerate the processing functions in hardware.

A hardware circuit can scale to process the increasingly complex rules at higher data rates. Hardware affords a higher degree of parallelism, which allows higher throughput to be maintained. A hardware device can be placed in line with a near zero increase in end-to-end latency. All network communication traverses through the

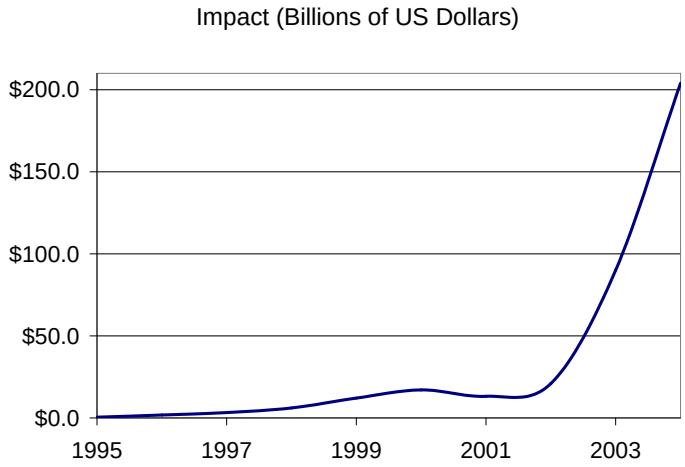

Figure 1.1: The cost of malicious code is increasing exponentially [1, 2, 123].

intrusion detection and prevention device so that as long as a rule for an intrusion has been written, the intrusion system discovers it. Only recently have components of intrusion systems been moved over to hardware to take advantage of parallelism, as discussed in detail in Chapter 2.

1.2 Intrusion Detection

Intrusion detection, in the general sense, identifies anomalous, inappropriate, or incorrect access to a system. There has been much work on defining the types of intrusions [9, 32, 34, 40, 80], distinguishing an intrusion from normal activity [16, 17, 57], and prototyping various intrusion systems [8, 31, 62, 70, 101, 105].

A high-level view of the components necessary to assemble an intrusion system is shown in Figure 1.2. At the center of the system is a component that detects intrusions. Four elements surround the detector that send and receive information. First, the detector has to know what events are classified as intrusions. When a new event occurs, the detector uses information about the current settings of the system as well as information about known intrusions to determine if this event is suspect. If the detector determines that the event is an intrusion, the event can be logged, a countermeasure can be taken, and an alarm can be raised. The potential countermeasures are represented as a database because multiple types of responses are available. An alarm could be signaled or the system could be modified to prevent similar events. When an alarm is triggered, an authority decides what further steps to take.

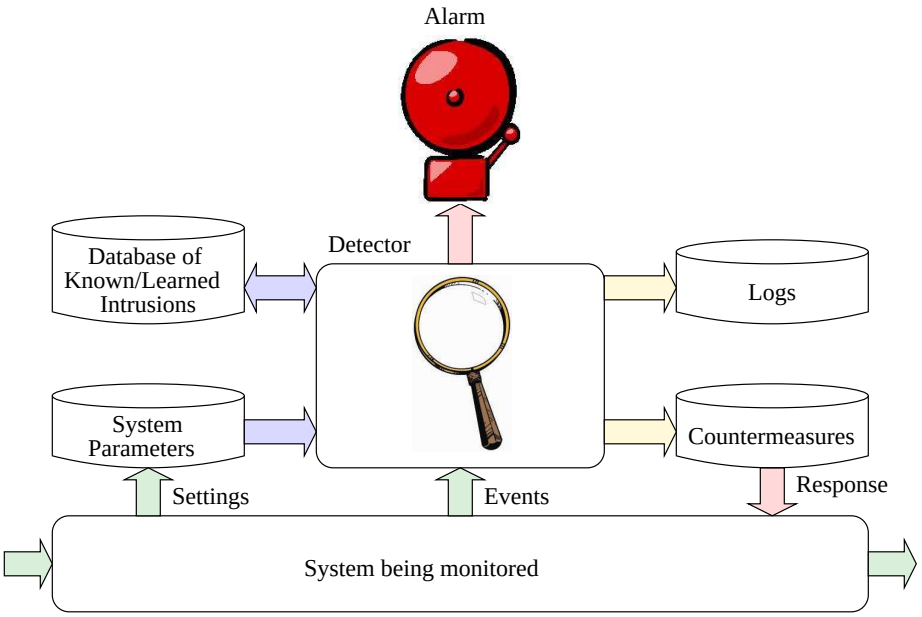

Figure 1.2: An intrusion system consists of a detector that recognizes known intrusions, learns new types of intrusions, and takes actions based on events that occur, raising an alarm if necessary. A similar diagram is given in [32].

Feedback from the detector to the database of known intrusions indicates that the ideal detector can discover new intrusions. The event may be an abnormal event or it may be patterned after a similar known intrusion. An authority can be consulted to determine whether the event is deemed an intrusion or not.

In the case of data networks, intrusion detection refers to the transfer of unwanted, malicious, or dangerous content over a network, and the system being monitored can be a web server, a database, or a cluster of computers. The intrusion may be as benign as spam or as harmful as a trojan horse [34] that infects a computer system by reading, writing, or even deleting files.

1.2.1 Types of Intrusions

Intrusions can take several forms. They can occur as abnormal, unauthorized, or unwanted system usage. Examples related to networking follow.

Unauthorized Access

Unauthorized access occurs when an individual gains access to a system they have no right to use. For example, a user may view web pages containing proprietary information that they have not been authorized to view.

Authorized Access

An intrusion can occur even if the credentials of the individual accessing the system are correct. For example, an intruder can fraudulently obtain account information such as login names and passwords. This is known as masquerading [16, 34, 40]. The system believes the intruder is authorized. This is the most difficult type of intrusion to detect since the detector must consider what is being accessed and what operations are being performed.

Spam

Spam is an unwanted electronic message from individuals or companies who send the message to people that may not desire to receive the message. These messages generally try to sell items, such as medication, loan applications, or pornography.

Phishing is a heinous form of spam where a message supposedly from an authoritative institution, such as a bank, e-commerce site, or government agency, directs the recipient to reply to the message or go to a web page and enter sensitive information. These messages can be quite persuasive, claiming accounts will be deactivated unless information is verified. Institutions have been formed in order to combat this problem [4], and tools are needed to protect network users.

Virus

A virus is a piece of malware hidden in files or emails. Once activated by the host, the virus replicates itself and spreads to additional hosts. Viruses generally spread via email, requesting that the recipient view an attachment. Clever virus writers writer code to search an infected host's address book to find additional recipients. The virus assumes the host identity when sending new email messages, increasing the likelihood that the target becomes infected.

Worm

A worm exploits a vulnerability in a system to execute code without the user actively starting it. The most common form of worm exploits buffer overflows, whereby the processor stack is subverted. Malicious code extends beyond the allocated buffer and is executed.

Worms take advantage of software flaws that may be difficult to find but are quick to exploit. The authors of [85] predicted that a worm could be written that infects all vulnerable hosts in less than 15 minutes. The prediction became reality in January 2003 when the Slammer worm infected over 90% of the vulnerable hosts within the first ten minutes [74, 75].

Internet users have been lucky so far in that the worms released have not appeared to be exceedingly malicious. However, even a fast propagating benign worm has the consequence of clogging networks with a flood of traffic. The authors of [76] developed guidelines to help stop the spread of such malicious code.

Denial of Service

Denial of Service (DoS) prevents legitimate users from accessing a system. DoS is accomplished by flooding a system with data that takes time to process. This inundation of events grinds services provided by the system to a standstill as each request is processed sequentially.

Web servers and email servers are frequent targets of such attacks, and the effects of a DoS attack can be very detrimental to those providing the service. E-commerce is especially sensitive to such attacks, since any loss of service can mean the loss of a customer's business.

1.2.2 Methods of the Detection

There are three basic ways to detect an intrusion: anomaly detection, signature detection, and learning. The detector should signal an alarm when a breach of security is attempted.

Anomaly Detection

In anomaly detection, the detector recognizes deviations from standard behavior. Abnormal behavior is considered suspect. For example, a flood of traffic to a particular

TCP port could signal the beginning of a worm/virus outbreak or a DoS attack. However, anomaly detection can result in false alarms. The event in question may be a previously unseen event that is perfectly legitimate, such as the distribution of a software update. The use of thresholds and statistical analysis is used heavily to prevent false alarms [43, 55, 71, 72, 116].

Signature Detection

The second method of detection is to search packets or flows for known signatures. This requires that the detector know what to search for in advance. This could involve searching packet headers for suspect port numbers or IP addresses, or it could trigger searching payload for worm or virus signatures.

Learning

Finally, an effective detector should be able to learn about and react to new intrusion attempts. Learning requires training time for the system to determine what constitutes normal behavior and who are legitimate users [59]. Once trained, the system reacts to abnormal behavior and makes decisions on what actions to take.

One technique to learn about new signatures is to use thresholds to determine if a signature appears to be occurring too frequently. The authors of [71] used thresholds in conjunction with a timeout period to detect suspect occurrences of signatures.

1.2.3 Responses

Once an intrusion is detected, a response should be taken. The response could be to write to a log file or to email an administrator. The type of response depends on the way the system is configured. An intrusion detection and prevention system (IDPS) can be configured as either passive or active.

If the system is configured to be passive, as intrusion detection systems are, countermeasures cannot be performed to stop the intrusion because the system is only being monitored. Passive systems just inform an authority of security breaches. It is left to that authority to determine what to do about the problem.

Intrusion prevention systems can take countermeasures. In-line, active systems stop the flood of worms, which otherwise infect all vulnerable hosts in a matter of minutes. Countermeasures may include dropping an offending packet, terminating a user's connection, or blacklisting an IP or email address.

Effects of False Alarms

False alarms in intrusion detection systems are a serious problem. A false alarm occurs when an event or sequence of events causes an alarm to trigger even though the event was legitimate. For example, a false alarm may be raised at a web server if a certain web page becomes popular. A flash crowd, for example, occurs when there is a sudden surge of interest in a particular page. A term called "slashdotted" has been coined to describe the effect of large scale access to a web page when the URL is posted in an article on the `www.slashdot.org` website. An intrusion detection system (IDS) may determine this is a distributed DoS attack when, in fact, the web server cannot meet the demand of all the legitimate users who want to access the content.

When an alarm is raised, the reason for the alarm should be genuine. A system that generates many false alarms results in real alarms being overlooked. Additionally, when events occur too often, the logging and alarming mechanisms become overloaded. The authors of [12] and [90] discussed methodologies for benchmarking intrusion detection systems.

1.2.4 Who is the Authority?

Intrusion detection systems are used by many different types of people, ranging from individuals on home computers to the staff at large corporations. The focus of this work is for systems operating on large data networks, including local area networks (LANs) and wide are networks (WANs).

Members of an Information Technology (IT) department are generally granted authority to access networked systems, enforce networking policies, ensure the network is properly maintained, install new system software, and ensure the data within the network is only accessible by those with proper credentials. When an intrusion is detected, they respond and make the decisions about how to combat the intrusion.

1.3 Rule Processing for Intrusion Detection

Rule processing enables an authority to describe what constitutes an intrusion in his or her network. Rules can be written to perform header processing and/or payload signature detection. The rules are then used by the detector to scan events as they occur in the system being monitored.

Rule processing requires many resources and much computing power. Anomaly detection and learning algorithms could be used in tandem with rule processing systems by defining new intrusion rules.

Writing useful intrusion rules requires care. A poorly written rule can result in the generation of a large number of false alarms. New rules should be created so they uniquely define an intrusion. The architectures to be described report rule matches when the criteria specified in the rule is detected. Whether the circumstances surrounding the match actually represent an intrusion is left as future work.

1.3.1 Necessary Characteristics

To be an effective intrusion detection and prevention system, several key characteristics must be present. The system must operate in real-time, rather than using off-line processing that could occur hours after the malicious content has infected the network. This point implies that the IDPS must operate at network link rates.

The system must be highly adaptable in order to create a defense against malware [130]. Rule updates must occur quickly and be put into effect with minimal overhead. Taking the system down to change the rules is not an option. Forms of intrusion are continually changing, and systems must be flexible to counteract new security issues.

The system must also be efficient. It must concisely inform network administrators of what is occurring in the network. Ideally, the system reports enough information so future intrusions are blocked. For instance, future traffic from the same source could be dropped or given a lower priority. Additionally, a full record of the events leading up to the intrusion and the occurrence of the intrusion should be logged so that an authority can examine the circumstances surrounding the breach.

1.4 Thesis Objectives

This thesis set out to explore rule processing architectures to perform intrusion detection and prevention for networked systems with the following goals in mind:

- Provide a scalable architecture that supports processing of databases that contain a complete set of IDPS rules
- Provide mechanisms that do not disrupt normal traffic but block malicious attacks

- Provide an intuitive interface for modifying rules and processing alerts
- Provide an useful tool for network administrators
- Be implementable in a form factor that is cost-effective and space-efficient

If this system is to be placed between systems, it should operate in a non-intrusive way so as not to damage communication between the end systems.

If a tool is to be adopted, it should be intuitive, well documented, and serve an useful purpose. It should provide easy mechanisms to add or remove rules to the system and coherently report alarms to an authority.

Finally, the cost of the system should be considered. Current software-based intrusion systems can require racks of computers to monitor a connection [54]. If this functionality can be reduced to a single device, numerous benefits in terms of reducing power, space, and cooling can be achieved.

1.5 Contributions

This thesis presents three separate architectures to perform rule processing for network intrusion detection systems (NIDS) in reconfigurable hardware. The primary contributions of the work are:

- The development of a scalable rule processing framework for intrusion detection.
- The comparison of different approaches to rule processing.
- The development of reconfigurable modules that can be reused. Components that include Bloom filter hardware, a communication wrapper, and a statistics engine were developed that can be re-targeted for new projects.
- The analysis of real network traffic to determine the frequency of rule matches, benefiting architects in optimizing designs for actual network systems.

1.6 Thesis Outline

This thesis explores architectures to perform rule processing, laying a foundation for an efficient intrusion detection and prevention system in reconfigurable hardware.

In Chapter 2, background information on processing technologies and networking protocols is provided. Snort, the most common IDS currently available, is discussed, with a focus on the algorithms used and the difficulties of directly porting Snort into hardware. Using Snort rules, the elements involved in rule processing are discussed: flow reconstruction, header processing, and pattern-matching. Related work is also provided.

Snort Lite is introduced in Chapter 3. This architecture implemented a subset of Snort rules on a single FPGA device. The complexity of the problem forced a number of simplifying assumptions. The design decisions are discussed, as well as some key observations that aid the design of the other two architectures. This architecture is considered the baseline implementation of an IDPS system.

In Chapter 4, the Snort Intrusion Filter for TCP (SIFT) architecture is presented. This architecture explored a hardware and software approach to rule processing. In SIFT, questionable data is sent to software for further processing. Questionable data is determined by a combination of header processing and content scanning. Only packets with a matching header or a keyword in the payload are forwarded to software. A key observation about rule match frequency was learned from this architecture.

In Chapter 5, the rule processing framework is described. This framework provided the most flexibility and the greatest amount of scalability. The architecture divided the header processing and content-scanning into separate modules and relied on an adjacent hardware processing circuit, called the rule processor, to determine whether intrusion rules matched.

In Chapter 6, system analysis and testing results are presented. Rather than present the individual testing results for each of the architectures in their respective chapters, these aspects are presented together here. All systems have their own unique benefits and excel in different ways.

In Chapter 7, the work is summarized. A discussion of potential future work in this area is provided.

The appendices are provided for reference. Appendix A discusses the control software for the three systems. Appendix B discusses the laboratory configuration to test each architecture. Appendix C explains the source code directory structure and how to use the provided scripts. The location of the source files is also given. Appendix D gives additional figures associated with system testing. Appendix E provides a list of acronyms used throughout this thesis.

Chapter 2

Background

In order to present rule processing architectures, a background on related work is needed. First, an overview of the technologies available to perform rule processing is given. Second, a brief description of the main networking protocols used by the system is given. Third, current rule processing systems are presented. Fourth, Snort is examined in detail, focusing on the underlying mechanics and its limitations. The rules currently found in Snort are analyzed, giving careful attention to rule characteristics. In the remaining three sections, details about the three main aspects of rule processing are given: TCP flow reconstruction, header processing, and string matching. Relevant related work is presented in the appropriate sections.

2.1 Implementation Medium

Techniques to perform rule processing can be implemented on a variety of mediums, from general purpose processors to custom hardware.

2.1.1 General Purpose Processors

The general purpose processor (GPP) offers the most flexibility to implement features for rule processing. Through the use of network interface cards (NICs), an ordinary PC can become a NIDS. Software exists today that can perform the tasks of rule processing. However, what advantage the PC gains in flexibility, it loses in raw computing power.

2.1.2 Network Processors

Network processors, such as the Intel IXP 2800 [52], are built specifically for networking applications. A standard programming language allows the application designer to perform operations directly on network data at speeds much greater than a standard PC. This allows for relatively quick implementations.

The IXP 2800 has 16 parallel microengine processors, each of which can support up to eight processing threads. With large numbers of fast general purpose registers and multiple interfaces to external memory, a network processor can sustain high processing rates.

2.1.3 Field Programmable Gate Array

Field programmable gate array (FPGA) technology enables a fully-custom, high-speed, and flexible hardware implementation of data processing circuits. FPGAs use configurable logic blocks (CLBs) that contain look-up tables (LUTs) and flip-flops. Boolean logic functions are implemented using the CLBs. The speed of a design is determined by the location of the CLBs used and the routing delays incurred by connecting them. Fast, on-chip memories are also spread throughout current FPGAs.

By allowing soft modification of the hardware, the non recurring expense (NRE) of FPGA designs is lower than for ASIC designs. For low-volume applications or prototyping, FPGAs are the most cost-effective solution. The design cycle for a FPGA is considerably shorter than a fully-custom hardware solution. A developer writes VHDL [11, 126] or Verilog descriptions of hardware circuits, compiles the design for simulation, synthesizes the design, and finally maps the design into the appropriate part.

The architectures to be described use FPGA technology. Low-cost prototyping and reconfigurability are key factors for this decision.

2.1.4 Application Specific Integrated Circuit

An application specific integrated circuit (ASIC) is a fully-custom circuit that provides a low-power, high-speed solution to perform rule processing. However, due to the rigidity of the medium's fabrication method, once the design has returned from a fabrication plant, changes cannot be made without re-fabricating another circuit. This is not desirable for rule processing IDS since system requirements frequently

change. The NRE associated with an ASIC design is prohibitive for an application that needs to adapt continually. ASIC designs are generally only profitable for applications that produce hundreds of thousands of units.

2.2 Networking Protocols

The Internet can be described using a seven-layer model [86]. Starting at the lowest level, the layers are physical, data link, network, transport, session, presentation, and application. The physical layer is concerned with how bits of information are transferred from one location to another. The data link layer defines frame formats to specify where information begins and ends. The network layer defines how data is forwarded between hosts. The transport layer defines how data is transferred reliably. The session layer determines how communication sessions are created and authenticated. The presentation layer defines how data is internally represented for transmission. Finally, the application layer generates and/or interprets the data that has been transferred [56, 86, 117]. Rule processing resides in layers three and above.

2.2.1 Asynchronous Transfer Mode

The asynchronous transfer mode (ATM) protocol consists of 53-byte cells containing a virtual path identifier (VPI), virtual circuit identifier (VCI), header error correct code (HEC), and 48 bytes of payload data. The VPI and VCI are used to forward cells between destinations. An ATM network requires connections to be pre-allocated before data can be transferred. An allocation consists of configuring all VPI/VCI routing tables between the source and destination. The VPI field is used to route groups of VCIs together [86]. IP packets are transferred over ATM using specific protocols [48, 60].

2.2.2 Internet Protocol

The Internet Protocol (IP) is used extensively today in the global network, providing a best-effort delivery of IP packets [87]. A typical IP packet header consists of 20 bytes, as shown in Figure 2.1. The main fields of the header are:

- ToS - type of service, used for applications requiring certain quality of service (QoS) guarantees

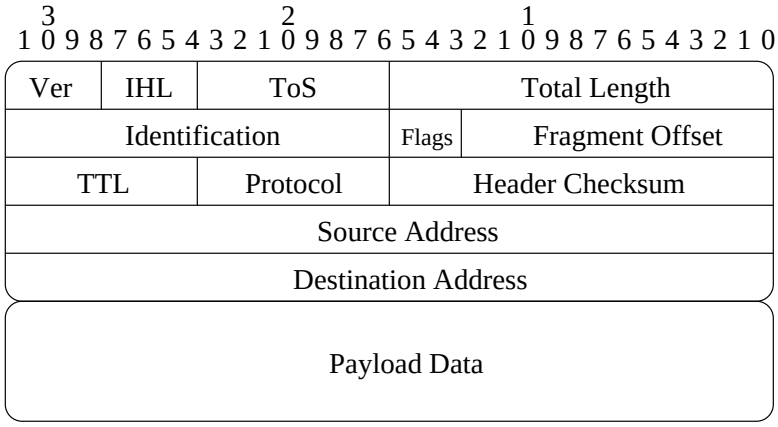

Figure 2.1: A typical IP packet consists of 20 bytes of header and up to 1480 bytes of payload data on an Ethernet network.

- Total length - the length, in bytes, of the entire IP packet
- TTL - time to live, the maximum number of hops the IP packet can make in the network before being discarded
- Protocol - the encapsulated protocol used in the IP packet
- Source Address - the source network and local address of the sender
- Destination Address - the destination network and local address of the receiver

IP packets are the fundamental unit of processing for the rule processing architectures. Higher level protocols, such as the User Datagram Protocol and the Transmission Control Protocol are encapsulated within the IP payload data.

2.2.3 User Datagram Protocol

The User Datagram Protocol (UDP) is a best-effort delivery protocol [89]. UDP is most commonly used for multimedia applications such as streaming video and audio. The main addition of UDP over IP is port numbers, which allows the operating system to deliver data to the appropriate application.

2.2.4 Transmission Control Protocol

The Transmission Control Protocol (TCP) is the predominate protocol used today [88]. The authors of [102] showed 85% of network traffic is TCP. TCP provides

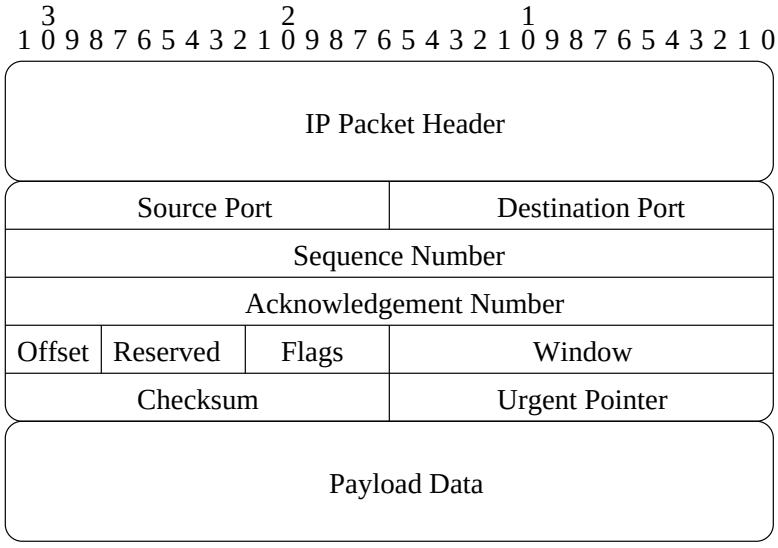

Figure 2.2: A TCP packet consists of 40 bytes of header information (20 from the IP packet header and 20 from the TCP header) and up to 1460 bytes of payload data on an Ethernet network.

a reliable, in-order transmission of data. While protocols such as IP and UDP are stateless, TCP is a state-based protocol, requiring a connection to be established. By maintaining state, large transfers of data are possible. The principle application of TCP is reliable data transfer, such as for web page viewing, email, and file transfer [29].

A TCP packet, as shown in Figure 2.2, appends 20 additional bytes of header onto the IP packet header. Fields of note are:

- Source Port & Destination Port - numbers to aid the operating system in determining where to send the payload
- Sequence Number - the number given to the first byte of data found in the payload to properly order data for delivery to the application
- Acknowledgement Number - the number given to the next byte expected at the receiver, which informs the sender as to what bytes have been received
- Window - the number of bytes that can be in-flight between the sender and receiver

TCP data is transferred in flows. A flow is characterized by four fields: the source IP address, the destination IP address, the source port and the destination

port. These four fields uniquely identify a TCP communication channel between the sender and receiver. Since the maximum transfer unit of most networks is 1500 bytes (Ethernet frames) and most files are larger than 1460 bytes, a flow needs to be established in order to reliably transfer all data bytes in the file.

2.3 Current Rule Processing Systems

Several rule processing systems have emerged as intrusion detection systems have been developed. Current implementations support certain aspects of rule processing, while neglecting others. Firewalls and policy engines are the two forms of rule processors.

2.3.1 Components of Rule Processing

The first major aspect of rule processing is flow reconstruction, which involves ordering TCP flow data. The best-effort nature of Internet routing protocols provides no guarantees that TCP packets will arrive in-order or at all. For this reason, a mechanism to handle TCP retransmissions is necessary.

The second main aspect of rule processing is header processing. All matching header rules need to be considering when performing rule processing. Fields in packet headers are inspected for known values that signify questionable material. For example, some viruses spread on certain port ranges. Rules can be written to raise alerts when these ports are used.

The final aspect of rule processing is pattern matching. The payload of packets is examined for known signatures, such as an unique worm or virus signature or a watermark. In the case of TCP flows, this means examining the data stream that can span multiple packets.

Rule processing systems have only recently begun to incorporate the three aspects of rule processing. Snort, in its default setting, monitors the network on a packet by packet basis. Before the architectures developed for this thesis, there were no known hardware implementations that incorporated all three aspects.

2.3.2 Firewalls

A firewall is a device that inspects packets before they arrive at their destination. If the packets are found to contain questionable data, they are flagged. Firewalls can be used to drop traffic entering a network, or they can be used to prevent traffic from

leaving a network. The term "firewall" is used to suggest the prevention of spreading harmful materials from one area to another.

The earliest firewalls were based solely on examining the header of IP packets [58, 91]. These implementations relied on allowing known port numbers and IP addresses to pass through. For example, TCP traffic destined to port 25 or port 80 is generally safe since these are the ports for email and web traffic. However, an attacker can easily hide intrusions in these well-known port numbers.

Header-based software solutions are still common, and can be effective at removing a significant portion of unwanted traffic. *Zone Alarm* is a common example of a firewall solution.

2.3.3 Policy Engines

A policy engine examines more areas of a packet than just the header before deciding whether a packet is safe or not. The problem with current policy engines is their complexity. As a result, they passively monitor the network. Policy engines perform signature detection, correlate events, and compute complex logical operations.

Paxson et. al developed an IDS called Bro [84, 106]. Using a proprietary security language, this software-based system used *libpcap* to read network packets on a PC. Event engines used *libpcap* to validate packets, correlate the received packet with similar packets from the same flow, and process payload data. If alerts were generated, a policy script was run to determine what action to take.

Ilgun and Kemmerer used state transition analysis to perform rule processing [50, 51]. The concept is similar to finite automaton approaches used for string-matching and the Aho-Corasick algorithm [7]. When an event occurs, the current state of the system changes from secured to compromised.

Snort is another type of policy engine that uses rules to determine whether intrusions have occurred [3, 93]. Snort has been adopted as the tool for intrusion detection.

2.4 Snort: A Detailed Look

Snort is a software tool that was developed to limit high system footprint, simplify deployment, and reduce cost [93]. In the remainder of this section, the configuration of Snort is discussed, examples of Snort rules and their characteristics are given, the

internals of how Snort operates are shown, and the difficulties of porting Snort to hardware are described.

2.4.1 Setup

Snort is intended to be used in conjunction with a firewall system. Sensors are placed before and after a firewall [100]. Snort is run on several IDS computers using *libpcap* to read packets from the network being monitored.

2.4.2 Snort Rule Set

Intrusion rules specify: the processing of packet headers, the matching of patterns in packet payloads, and the action to take when a rule matches. A Snort rule is given below, and would cause an alarm for the packet shown in Figure 2.3.

> 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 must be TCP. If the signature was found in an UDP packet, it is not a match. The second part of the rule specifies to search for *"I Love You"* over an established TCP/IP connection. Flow reconstruction from multiple packets must be performed since there is no guarantee that the signature will not be segmented across multiple smaller packets. All of the tasks described above must be performed just to process this single rule. There are currently 2,464 rules in the Snort database. Over 80% of the rules require performing steps like those in the rule above.

In general, rules take the form:

$$< action > \; H1_{ID} \wedge H2_{ID} \wedge Hn_{ID} \wedge (S1_{ID} \wedge S2_{ID} \wedge \cdots \wedge Sn_{ID});$$

Note that multiple headers and multiple strings can be associated with a rule. The Snort database does not allow the logical *OR* in a rule. To perform this, separate rules are used.

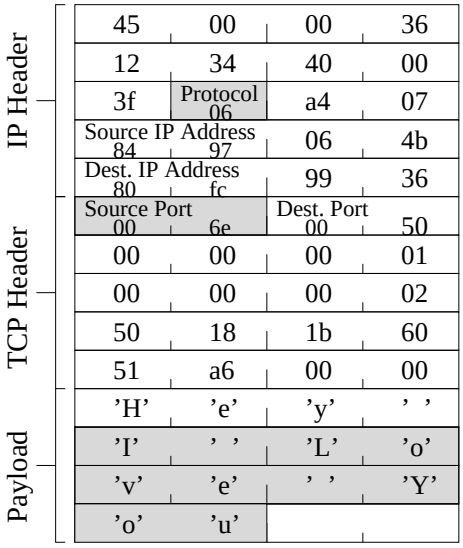

Figure 2.3: A TCP/IP packet that matches the rule specified in the example. The matching protocol, source port, and payload are highlighted. Four bytes are shown per line, and the bytes are shown in mixed ASCII and hexadecimal.

Characteristics

Before setting out to build any rule processing system, knowledge of the characteristics of rules is necessary. 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.

There are 292 unique header rules, 2,107 unique static signatures, and 233 regular expressions in the Snort rules database from September of 2004 (version 2.2). Most Snort rules specify header rules of the form external network to internal network. The signatures are distributed across the range 1 to 122 bytes, as shown in Figure 2.4. The bulk of the distribution is below 40 bytes. All 2,107 signatures are spread across 2,296 of the 2,464 rules. Figure 2.5 gives a histogram of the occurrences of signatures in 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 2,107 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.

2.4.3 Rule Features

Table 2.1 is divided into header and payload options that are available to the intrusion rule writer. The header options are split into sections that represent where they are found in a packet, starting with the IP header, then the TCP header, and finally the ICMP header.

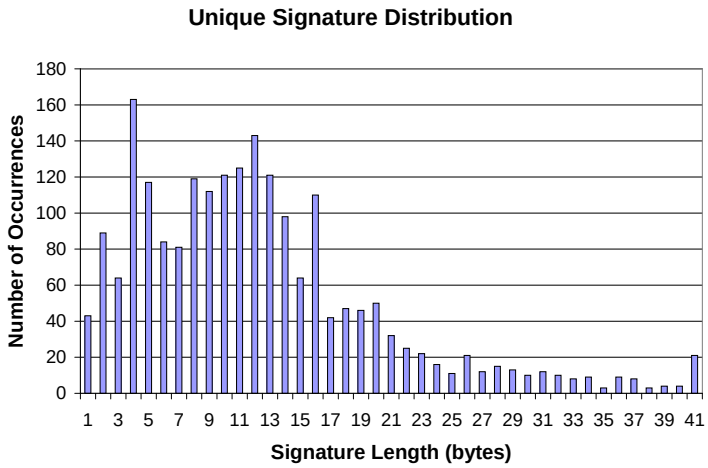

Figure 2.4: The number of signatures associated with each length.

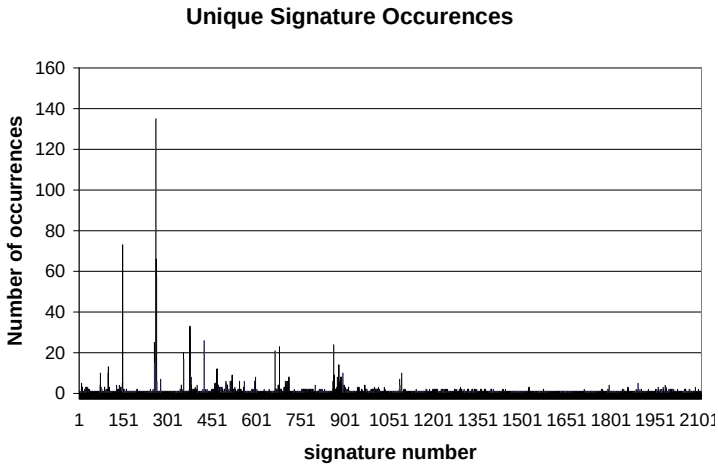

Figure 2.5: The number of times a signature appears in different rules.

Header Options

The header options shown in Table 2.1 correspond to distinct fields in packet headers. The IP addresses and ports are unique in that they allow ranges and masks. The other fields are exact match. As an example, consider the header rule below:

Table 2.1: The header and payload options that are available to Snort rule writers.

Header Options	Payload Options
Protocol	Content
IP Addresses	Perl compatible Regular Expressions
Same IP	Uricontent
TTL	Case Sensititivity
ToS	Offset
Identification	Depth
IP Options	Within
Fragment Bits	Raw Bytes
Fragment Offset	Byte Jump
Data Size	Byte Tests
Ports	
Flags	
Sequence Number	
Acknowledgement Number	
Flags	
ICMP Type	
ICMP Code	
ICMP Identification	
ICMP Sequence Number	

alert tcp 128.252.0.0/16 :1023 → any !80:100 ttl:32 ack:12500

This rule states that a TCP packet with a source network address of 128.252, a source port less than or equal to 1023, any destination IP address, a destination port that is *not* between 80 and 100, a time-to-live value of 32, and an acknowledgement number of 12,500 is considered a match.

Payload Options

Payload options are concerned with the presence and location of strings to find expressed as either static signatures or regular expressions. The *depth* construct allows the search of a specified string up to a certain location in the payload. The *offset* construct specifies where to begin looking for a given string. As with the header options, payload options can be mixed together, and multiple signatures can be specified in a rule.

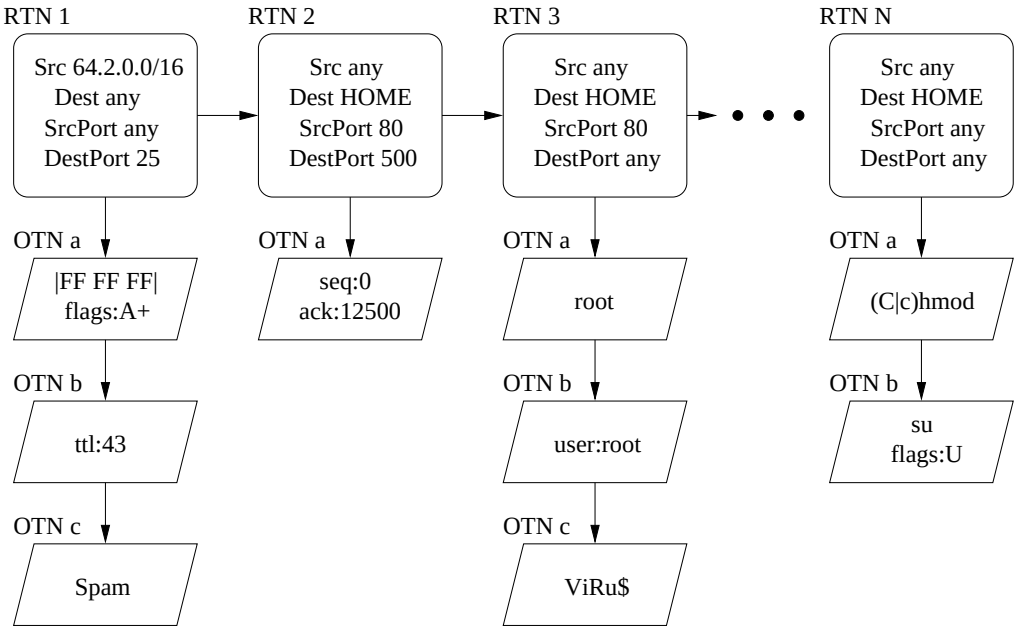

Figure 2.6: Snort rule processors consist of two dimensional trees that are separated by protocol. The IP address and ports are specified in RTNs. All other options are specified in OTNs.

2.4.4 Algorithms

When Snort receives a packet, it first splits up the processing based on the protocol of the packet [35]. For each protocol, a rule tree exists containing rule tree nodes (RTNs) and optional tree nodes (OTNs), as shown in Figure 2.6. There are two primary ways the string matching functionality is performed in the OTNs: Boyer-Moore or a modified Aho-Corasick.

When a RTN matches, the list of OTNs linked from it are also checked. For example, consider an incoming TCP packet with a source port of 80 and a destination port of 500. In this case, the OTN containing a check for sequence number 0 and acknowledgement number 12,500 is performed. If the sequence and acknowledgement numbers match, no further RTNs are examined because Snort only returns the first matching rule.

Boyer-Moore

The Boyer-Moore method for pattern matching is a shift-and-compare algorithm [22]. Consider searching for the term *String* in the sentence *We are looking for the word String*. The example is shown in Figure 2.7a through Figure 2.7c.

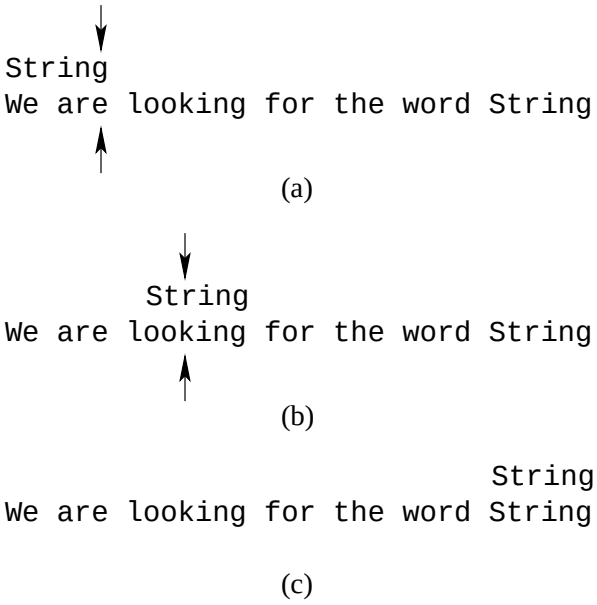

Figure 2.7: Character comparison begins from the right end of the word.

The algorithm begins in Figure 2.7a by aligning *String* along the left edge of the data to search through. Beginning at the end of the search term, it compares g to e and finds that they are not equivalent. The algorithm preprocesses the set of search criteria to determine how many characters *String* can be advanced. In the worst case, *String* is advanced one character at a time.

In Figure 2.7b, the algorithm has advanced *String* to be above the word *looking*. In this case, the algorithm successfully matches the suffix *ing*. However, the fourth character from the right is a mismatch (r and k). The algorithm searches for the next occurrence of *ing* to align the term *String*.

Finally, in Figure 2.7c, the search term *String* is found in the sentence. A match is then reported.

Aho-Corasick

The Aho-Corasick method represents search criteria in finite state machine (FSM) form [7]. Data to be searched is shifted through one character at time. For example, consider two search terms: *String* and *ran*. The state machine generated would look like that of Figure 2.8.

Ten states are generated. The state represented by digit 0 is considered an idle state. As characters are shifted through, the transition conditions are examined. If they are met, the FSM advances to a subsequent state. For example, assume the state

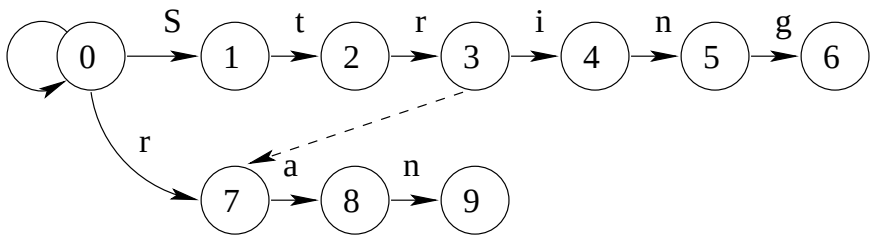

Figure 2.8: State transitions are performed based on the incoming character being compared. The failure function is shown as a dashed line.

is currently 3 because *Str* was seen. However, the next character is *a*. In this case, the algorithm needs to use the failure function to recognize that *r* exists in another path, one where the letter *a* can allow it to advance state.

Snort uses a modified version of Aho-Corasick that examines multiple characters to transition from one state to another. This eliminates many unnecessary comparisons [47, 121].

2.4.5 Performance

Since Snort has become the de facto standard for NIDS, a number of groups have worked to measure the performance of the system. As the number of header rules and signatures to match increases, the number of packets dropped by the sensor also increases. It is unacceptable for an IDS to not examine some packets.

Schaelicke et. al found that Snort inadequately acts as a sensor on higher speed links [94]. Their study showed that Snort alone is not to blame, but the platform running the software is partially responsible. Architectural decisions and the memory subsystem are critical factors in the performance of the NIDS. They found that even on a dual Pentium-4 Xeon running at 2.4 GHz with Hyperthreading technology, the system could only support 543 rules in the best case such that no packets were ever dropped. Furthermore, the authors found that only two of their test systems could support saturated 100 Mbps links. This is troublesome because Gigabit links are common today.

Lee et. al described how coverage, cost, and resilience to system attacks must be considered when deploying an intrusion detection system [61]. They used Snort with a subset of the available rules to show how it could be overloaded. In their experiments, Snort was overloaded when traffic exceeded 40 Mbps. During the periods of packet loss, Snort was only able to find 10% of generated attacks.

String matching is a very expensive task and was found by [10] to be at least 31% of the entire computation time and upwards of 80% of the processing time for web traffic. Verifying checksums found in the IP and TCP header is also an expensive operation. The IP checksum alone requires a series of computations over the entire payload. Finally, physically transferring the packet from the wire to system memory is time consuming. The *libpcap* function used by Snort is not designed for high-speed capture [35].

2.4.6 Portability Difficulties

A brute-force translation of Snort from a software implementation to a hardware implementation is inefficient. The software implementation is inherently sequential, while hardware is efficient at implementing parallelism.

The features available are difficult to port to hardware. For example, performing string matching within certain bounds of the payload is a complicated task for hardware to perform due to the very specific requirements that can be placed on different strings.

There are three challenges to performing rule processing in hardware:

- Scalability to process and store increasingly complex rules
- Correlation between header classification and payload content
- Adaptability to changing environment

One of the most challenging tasks in a rule processor is correlation of criteria. Every packet can contain matches for multiple header classifications and payload signatures. The system must correlate these matches to determine rule matches. While a single rule is trivial to process, consider that there are 2,464 rules found in Snort. In software, the correlation is performed using linked lists in memory. Implementing the same lists in hardware is detrimental to performance due to the numerous memory look-ups required.

In order to protect against evolving threats, the system must be adaptable. Rules change over time as new threats emerge. The system must adapt to scan for new forms of malware [76, 85]. Reconfigurable hardware enables the system to adapt to new threats quickly and at low expense.

2.5 Flow Reconstruction

2.5.1 Requirements

Flow reconstruction is one of the three main aspects of rule processing. Since 85% of Internet traffic consists of TCP traffic [102], a mechanism to provide in-order delivery of streams to pattern matching circuits is necessary. To be a viable solution for rule processing, the solution must:

- Support a large number of simultaneous flows
- Maintain state for the current data window in all flows
- Handle retransmissions and redundant witnessed data
- Handle sequence gaps and unresolved data
- Support timeout of improperly terminated flows or excessively idle flows
- Validate packets are properly formatted
- Support non-stream-oriented protocols such as UDP
- Operate at high throughput

A flow reconstruction circuit must be able to handle a large number of simultaneous flows. Seeing as how most IDS sensors are placed at the gateway of a network, the number of concurrent connections can be hundreds of thousands. For each connection, state information is needed to maintain a consistent view of the bytes in the stream, what information is redundant, and what sequence-gaps have been opened.

Flow reconstruction is complex due to delivery mechanisms employed by IP networks. First, there is no guarantee that a sequence of packets will arrive at a given node in order. Second, there is no guarantee that all packets will arrive at a given node, either because of a dropped packet or the use of a different route through the network.

In addition to stream-oriented features, a flow reconstructor must also be able to perform the tasks associated with packet processing. This involves ensuring that packet checksums are correct and that packet lengths are accurate.

All of these functions must be performed at high throughput, limiting the amount of processing time available for each packet. Efficient implementations need to be capable of sustaining network line rates.

2.5.2 Techniques

TCP offload engines (TOEs) remove protocol stack processing from software [30]. A TOE processes the layers of TCP/IP stack such that the application need not perform protocol related computations, allowing it to focus on application-specific problems.

Nguyen, Zambreno, and Memik created hardware-based flow monitors [82]. Their flow monitor units provided components with flow-based information. They have demonstrated the ability to process very high packet rates for a limited number of TCP flows.

Necker, Contis, and Schimmel implemented a single TCP-stream assembler in FPGA technology capable of operating at 3.2 Gbps [81]. By using separate reassembly units, they estimated that up to 30 TCP flows could be tracked in a single FPGA.

Li, Torresen, and Soraasen implemented a state-based inspection technique for eight TCP flows in FPGAs that operated at 3 Gbps [64]. With a newer FPGA, they estimated that up to 70 simultaneous connections could be supported. The method was intended as an add-on for current software-based implementations of Snort.

The most dramatic results in this area are by Schuehler [96]. He developed a TCP Processor as a protocol processing wrapper implemented in FPGA logic that annotates control information onto incoming IP packets, specifying where headers begin and end and where payload data begins and ends [97, 98, 99]. The TCP Processor generates an ID for each TCP flow so that context switching can be performed by downstream modules that process TCP streams. The TCP Processor is capable of simultaneously keeping state for eight million TCP flows while operating at rates of 2.9 Gbps on the FPX platform. The author predicates that over 10 Gbps processing is possible through the use of faster FPGA devices. This technology is used in the architectures to be presented.

2.6 Header Rule Matching

2.6.1 Requirements

Header processing is the second main component needed for rule processing. As the name suggests, header processing looks at the fields inside packet headers, such as the IP addresses, protocol, and ports. In most case, header processing only needs to be performed once per flow. A flow is uniquely identified using the standard 5-tuple found in the packet classification literature: source IP address, destination IP

address, source port, destination port, and protocol. Rules that check other fields may be examined for every packet. In order to perform NID functionality, all matching header rules must be examined, not just the highest priority match as is the case in the longest prefix matching literature [33, 38, 118, 120, 124, 125].

2.6.2 Techniques

Three basic approaches to perform packet classification for rule processing have been developed: content addressable memories (CAMs), tries, and hierarchical grouping.

Content Addressable Memories

CAMs are high-capacity rule storage devices that compare all entries to the incoming packet header in parallel [73]. Ternary content addressable memories (TCAMs) extend the functionality of a CAM by allow masks to be incorporated in the comparison. A user can specify which bits of a field are important [83]. Due to the fact that a value and mask must be stored in a TCAM, the resource requirements are larger than that of a CAM.

Yu and Katz used TCAMs to return multiple matching packet headers for rule processing applications [128]. Gokhale et. al explicitly performed header processing and content matching using CAMs [44]. Lockwood et. al created a reconfigurable firewall that performed header processing using TCAMs implemented in FPGA hardware [68].

Song and Lockwood used a hybrid bit-vector and TCAM algorithm to compress the matching header representation [108]. The authors converted 222 Snort header rules into 264 trie-node prefixes and 33 distinct TCAM entries.

Tries

The trie (pronounced *try*) approach to packet classification involves creating tree structures, whereby matching values are represented by nodes in the trie. The trie is constructed using the upper, do-care bits of the IP address [113, 114].

The generic form of the trie examines one bit per node. However, this is inefficient, especially for IPv6 applications which use an 128-bit address instead of the 32-bit address used in IPv4. To combat this problem, a multi-bit trie is used. The multi-bit trie examines multiple bits per memory access. An example of both is shown in Figure 2.9.

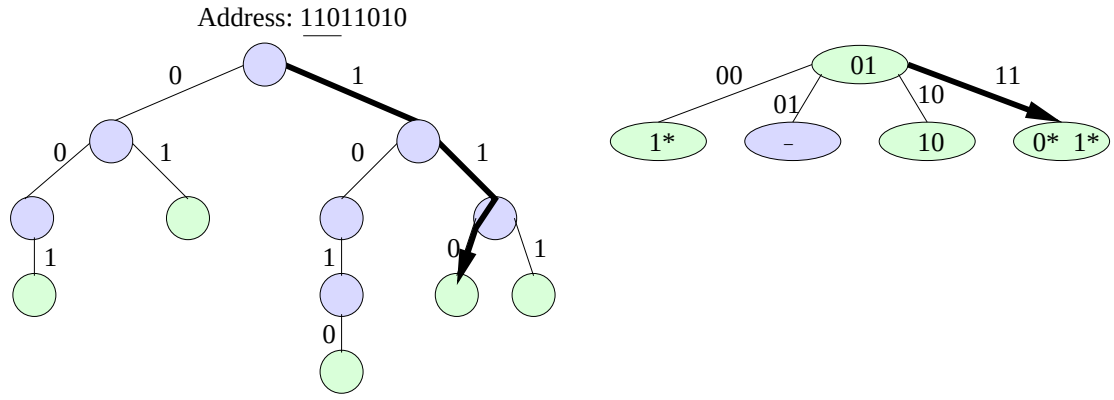

Figure 2.9: A single-bit trie and the corresponding multi-bit trie with a stride of two. Green nodes contain next hop routing information.

Hierarchical Grouping

Hierarchical grouping techniques preprocess the rule database, examining the rules to be used. Heuristics are used to determine which dimension to use to separate rules. Dimensions are determined by the fields that are being inspected, such as the IP addresses, ports, and protocol.

Gupta and McKeown generated *HiCuts* to group rules [45, 46]. This was later extended by Singh et. al in the creation of *HyperCuts* [104].

2.7 String Matching

String matching, sometimes referred to as pattern matching or signature matching, is the most computationally intense aspect of rule processing. Strings can appear anywhere within the payload of a packet or even span packet boundaries. This section focuses on hardware related techniques to perform string matching.

2.7.1 Requirements

String matching has two forms. The first form is static signature detection. This involves looking for an exact sequence of bytes in packet payloads. Static signature detection is used to detect viruses and worms.

The second form is regular expression detection. Regular expressions allow for patterns to be detected. Consider the regular expression *(R|r)eg(ular expression?|ex)*. This regular expression matches when any of the following signatures are found:

regular expression, *Regular expression*, *Regex*, or *regex*. Note that while this simple example could easily be searched for using four separate static signatures, regular expressions can be considerably more complex, subsuming hundreds of variations of the same basic sequence. For this reason, separate techniques are required for regular expression processing.

There are three requirements for string matching in hardware: high throughput, high capacity, and quick adaptation. As link speeds increase, the amount of data that needs to be processed grows enormously. Software is not a viable solution for Gigabit links or other high-speed backbone links. As more signatures are added to intrusion systems, the importance of high-throughput string matching becomes apparent [41].

The current Snort database contains 2,107 static signatures and 233 regular expressions. This amounts to over 30,000 characters. As new threats emerge, these numbers will increase. Therefore, the string matching techniques must be scalable to accommodate new signatures.

Finally, since signature databases are constantly being updated and revised, the ability to quickly incorporate new changes is imperative. With extremely harmful malware forecasted by [85] that is capable of infecting all vulnerable hosts within minutes, the need for quick adaptability is a must in order to stay the threat.

2.7.2 Techniques

There are many techniques to perform complex string matching through the use of FPGAs [42]. Network processors have also been used to perform fast string matching [65]. Current research focuses on efficiency, resource consumption, and module throughput. The approaches consist of four basic categories: automata, comparators, CAMs, and hashing.

Automata

The automata approach to string matching is to form a state machine to search for the string. The automata approach can be further divided into nondeterministic finite automata (NFA) and deterministic finite automata (DFA).

NFAs can be represented using a directed graph, where each node is considered to be a state and each transition is labeled with a character or ϵ, where ϵ is the empty

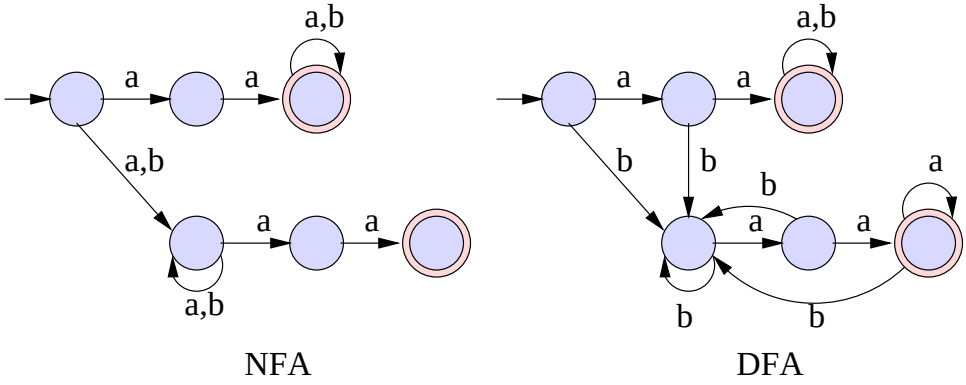

Figure 2.10: This example shows the NFA and DFA created for all strings with alphabet {a,b} that begin or end with *aa*. (Example courtesy of David Galles, University of San Francisco).

character. A regular expression of m characters maps to $O(m)$ nodes. If the pattern is searched for in a sequence of length n, the search time is $O(mn)$.

DFAs are similar to NFAs, but they do not contain the empty character in any of the state transitions. Additionally, no state is allowed to have more than one transition for a given character. A regular expression of m characters maps to $O(2^m)$ nodes in the worst case. The search time for a DFA is $O(n)$.

Consider the example of Figure 2.10. The example searches for all strings, using the alphabet {a,b}, that begin or end with the signature *aa*. Note that the NFA has considerably less transitions than the DFA implementation of the same search.

Sidhu and Prasanna started the recent work in the area of FPGA-based string matching by using NFAs that directly map into FPGA logic [103].

Clark and Schimmel reduced the redundancy inherent in NFAs [26]. Many signatures contain overlapping prefixes or suffixes. Used in a conjunction with their hardware platform [27], they used FPGAs to off-load the pattern matching from software. They have also shown in [28] how to expand to higher bandwidth nodes with multiple character decoders.

Moscola et. al created an automated way of generating DFA structures optimized with JLex, a lexical analyzer that maps regular expressions into Java code [66, 78, 79]. Circuits were generated from the optimized state machines, then mapped into the FPGA. It was found that the number of states necessary to implement the DFA was comparable or less than the number needed for NFAs in most cases.

Sugawara, Inaba, and Hiraki implemented a string-matching method using trie-based table-lookups in order to achieve high throughput [115]. Their system was also capable of operating on a limited number of TCP streams. The algorithm utilized a modified Aho-Corasick algorithm called suffix based traversing (SBT). State transitions were performed using table look-ups.

Comparators & Pipelines

The comparator and pipeline technique is a brute-force approach with optimizations. The signatures to scan are embedded in logic and compared to incoming input. Since comparisons are performed as data streams into the system, the comparisons are pipelined. That is, if the search term was *cat*, when pipeline stage three holds *c*, pipeline stage two holds *a*, and pipeline stage one holds *t*, a match is declared.

Baker and Prasanna implemented an efficient way of minimizing FPGA resources to perform pattern matching via comparators [18, 19, 20]. They partition signature databases into independent pipelines, allowing for FPGA resources to be efficiently utilized by reducing redundancy. Previous work by the authors created an improved version of the Knuth-Morris-Pratt (KMP) algorithm [53].

Cho, Navab, and Mangione-Smith created a content-based firewall using discrete logic filters [25]. They created automated techniques to generate highly parallel comparator structures for quick reconfiguration. The focus of the work was high-throughput, which was achieved by pipelining. They expanded their approach in [24] to include logic re-use and read only memory.

Sourdis and Pnevmatikatos created unique VHDL instances for each rule to process [110]. Rules are added or removed by modifying the instance loaded. They achieved high-throughput via deeply pipelined comparators and encoders and by reducing fan-out. The authors made further use of pre-decoding to raise the throughput in [111], reduce the redundant logic used, and fit more patterns in the FPGA.

Content Addressable Memories

CAMs and TCAMs have been used for string matching as well as header processing. The signature to match is placed in a CAM entry and is compared in parallel with all signatures currently in the database.

Yu and Katz formulated a way to overcome problems that arise due to the use of strict width requirements for TCAM entries [129]. Large signatures can be broken

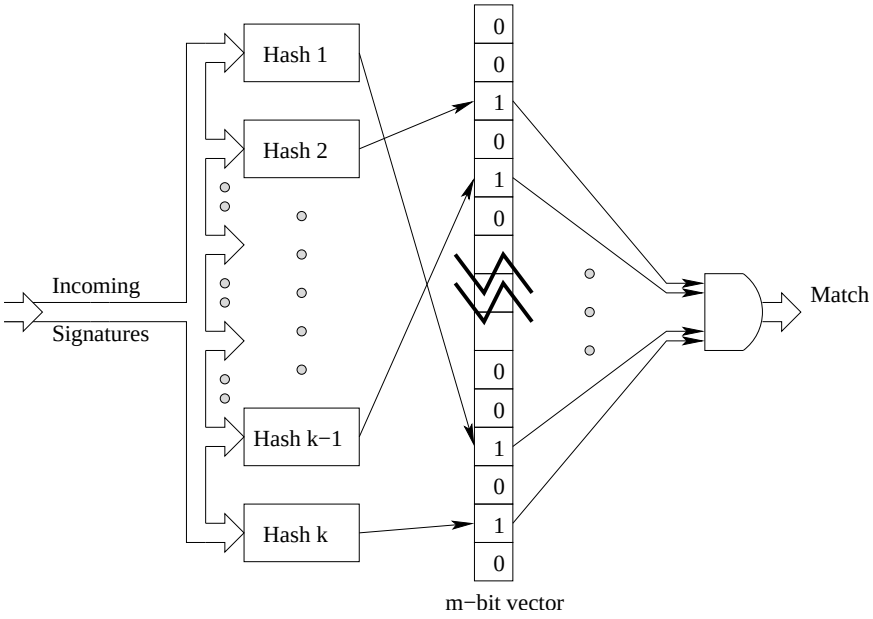

Figure 2.11: A Bloom filter consists of a vector of length m, k separate hashes, and an AND gate with k inputs.

across multiple entries. Correlation of many entries can be accomplished with outside logic. In their work, they show how to achieve Gigabit throughput.

Gokhale et. al investigated a hardware/software approach to intrusion detection, where a header and content vector of matches was sent to software for processing [44]. The system separated pattern matching and rule processing onto separate environments, the first in hardware and the latter in software. The system supported a few hundred rules. However, they were limited by the speed at which software can perform processing on returned match vectors. They hinted at an alternate version of the rule processor implemented in hardware that returned a bit-vector of rules. A rule matched if its corresponding bit is set in the vector.

Hashing

A more elegant approach to the problem of string matching is to employ hashing techniques. Bloom filters can be used to perform matching of large numbers of static strings [14, 21, 37]. The logic footprint is constant, regardless of the search criteria. However, a Bloom filter cannot search for regular expressions without expanding the regular expression into all possible static signatures. In most cases, all possibilities cannot efficiently fit into the Bloom filter.

A Bloom filter, as shown in Figure 2.11, consists of k hash functions, a m-bit vector, and an AND gate with k inputs. To determine whether a given signature exists in a search database, k hashes are computed over the signature, each indexing a particular bucket in the m-bit vector. If an '1' is found at each of the k locations in the vector, the input signature exists in the database of n signatures with a certain probability, as defined in Equation 2.1 [38].

$$f = (1 - (1 - \frac{1}{m})^{nk})^k \tag{2.1}$$

A false positive occurs when the k referenced locations in the bit-vector return '1', but the signature does not exist in the database. The value of k can also be interpreted as the number of bits required to store a signature. To add a signature to the bit vector, the same hashes that are used to query are used to set the appropriate bits in the vector. To remove a signature, the same hash functions are used to reset the bits if no other signatures reference the same location.

The false positive rate depends on the number of signatures that are loaded. Figure 2.12 shows how the false positive rate changes as n is varied for different values of k and m. As k increases for fixed m, the false positive rate shifts downward, meaning that there is a lower rate of false positives. This is due to the fact that more bits are being used to represent each signature. As m increases with k fixed, the false positive rate decreases because the space to distribute the signature has increased.

Bloom filters have the advantage of allowing for a large number of strings to simultaneously be scanned at high throughput. However, as will be explained in Chapter 3, the Bloom filter implementation can be tricky in order to remove false positives as well as efficiently scan for different length strings.

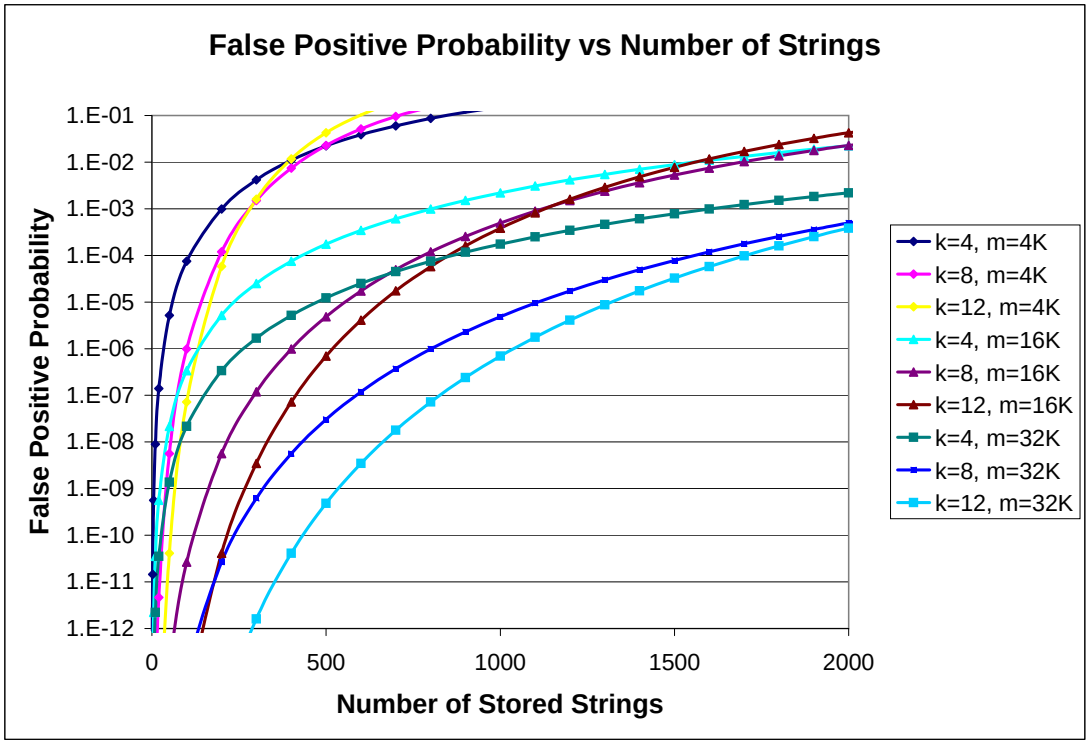

Figure 2.12: The expected false positive rate versus the number of signatures in the database.

Chapter 3

Snort Lite

Snort Lite is a light-weight rule-based processing system that scans Internet Protocol packets as they traverse through a network. This hardware-accelerated device implements network intrusion detection and prevention functions in a circuit that is in-line with a broadband network link. Packet header processing and content scanning circuits are compiled into hardware that can process intrusion rules in real-time. While similar software-based tools like Snort act as a passive monitor, Snort Lite can also filter and block malicious Internet traffic. Rules are added and deleted dynamically in order to adapt quickly to attack. Up to 18,400 signatures can simultaneously be searched.

In this chapter, the design objectives and design decisions of this work are discussed. An overview of the features supported by the system are given, and the architecture of the system is presented in detail. Finally, key items learned from the architecture are discussed.

3.1 Design Objectives

There were several goals for this project. The first goal was to show that signature detection and header processing can be implemented in a single FPGA. Using a single FPGA requires less space than a computing cluster [54] to perform similar IDS functionality. By using a reconfigurable hardware device, reduced power consumption and physical space can also be achieved.

Second, as link speeds increase, the need for a hardware solution to process and act upon packets in real-time is essential. An efficient software solution does not exist that can meet the requirements of NIDS, as shown in [61, 94]. By processing

traffic with a high throughput, the device operates in active mode, protecting against malware. This hardware solution processed data at 502 Mbps while inspecting all packets.

Finally, this design was intended as a baseline for future hardware-accelerated content processing systems. Questions about the size of input and output buffers, the locations of the critical paths, and how many resources to use were answered. Additionally, many of the components used in this circuit were designed in a generic, modular way to be re-used in other applications.

3.2 Design Decisions

3.2.1 Rule Types

To enable transition from software-based rule processing to hardware-based rule processing, the Snort rule syntax was adopted. Rules that consist of one header rule and one signature in the payload are considered a baseline rule, while rules that have more than one signature are split into multiple rules. Over 82% of the static signatures in Snort version 2.2 are used in a single rule.

Signatures scanned in the hardware circuit longer than 32 bytes are truncated. This reduced the logic required to perform the string matching. The current Snort database consists of signatures from 1 byte to 122 bytes. Of the 2,107 total unique signatures, 2,038 of them are less than or equal to 32 bytes.

3.2.2 Architectural Components

Bloom filters were used to perform string matching. Bloom filters allow large databases to be stored, and they have a quick update time, constant access latency, and a high degree of parallelism.

Bloom filters, when used alone, can report false positives. The hardware circuit implemented used an external memory to ensure no false positives were reported. Bloom filters scan a window of bytes for pre-defined lengths. The flow of bytes in the pipelined window dictate the throughput of the system. In this baseline design, the pipeline advanced one byte per clock cycle. Additional bytes could be shifted through per clock cycle by instantiating multiple Bloom engines in parallel at the cost of increased resource consumption.

A decision that needed to be made early on was how to communicate with the hardware device. Since the system processes IP packets, it was natural to communicate using UDP packets, allowing a PC to be used as a controller. Security concerns are addressed in Section 3.7.

3.3 Features

The following features were implemented:

- Header classification based on source IP address, source port, destination IP address, destination port, and protocol
- Support for network masks within fields so that bits can be selectively matched
- Support for port ranges so that numeric values in a range can be specified
- Support for up to 32,768 header rules
- Payload scanning for static signatures that can vary in length between 2 and 32 bytes
- Support for up to 18,400 static signatures
- Passive and active operation modes so that traffic can be inspected and/or dropped
- Support to forward the content of an offending packet to another machine for further inspection

Not all features found in Snort were implemented. Checks of parameters in the IP and TCP header fields for the type-of-service (ToS), time-to-live (TTL), sequence numbers, and flags were not implemented. While including these options makes the header processing complete, analysis of Snort rules showed that 95% were of the form *EXTERNAL_NET* to *HOME_NET* and did not include these aspects at all.

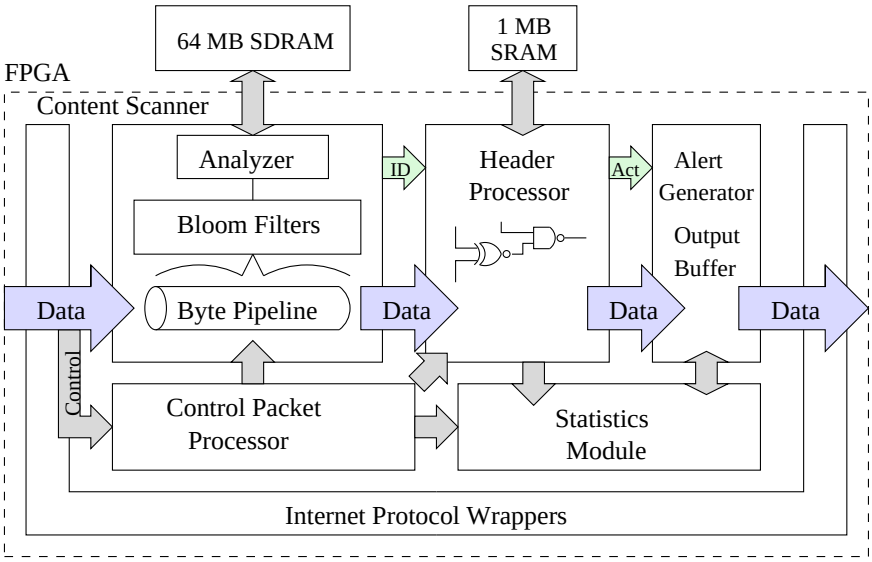

Figure 3.1: Snort Lite consists of a content scanner, a header processor, an alert generator, a control packet processor, a statistics module, and a set of Internet protocol wrappers. The entire design fits in a single Xilinx Virtex XCV2000E-8 FPGA and utilizes one external Zero-Bus Turnaround (ZBT) SRAM and one external SDRAM.

3.4 Architecture

3.4.1 Overview

The architecture developed for this project performs all rule processing in a single FPGA, and actions are taken based on the results of the rule processing. This system achieved greater volume of rules than software can process. However, the rules implemented are not as complex as what can be done in software.

The design of the Snort Lite was modularized to ease design and debugging. There are five main components in the design, as shown in the block diagram of Figure 3.1. Data flows through the Internet protocol wrappers to the content scanner. The packet data is funnelled through a byte pipeline that is monitored by Bloom filters to search for content. If a Bloom filter signals a match, an analyzer is queried to ensure the signature was a true match. If the signature is a true match, the analyzer returns the corresponding ID number of the signature to a header processor. The header information associated with the given signature is retrieved from SRAM, and the packet header is compared with the stored header entry. If the header matches, an alert message is sent to a software controller and the appropriate action is taken. The following sections describe each of the major components.

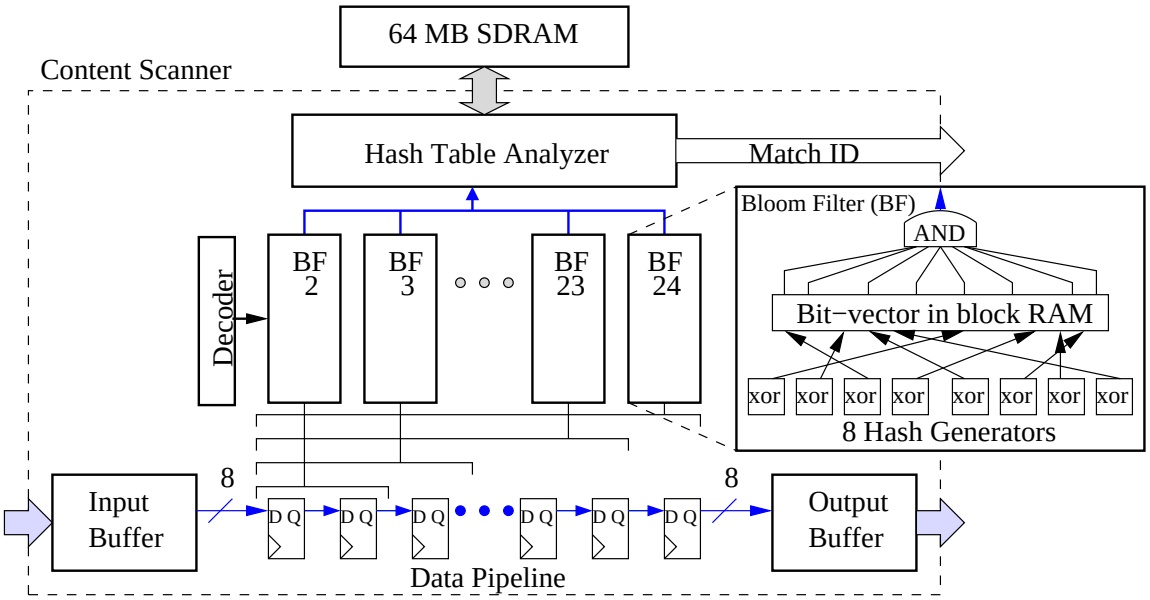

Figure 3.2: The content scanner consists of an input buffer, several Bloom filters (BF), a hash table analyzer, an output buffer, and a decoder. The data pipeline advances by one byte per clock cycle.

3.4.2 Content Scanner

The content scanner consists of five main components, as shown in Figure 3.2. Data enters the input buffer 32 bits per clock cycle. The input buffer converts the word-stream into a byte stream. This byte stream passes by 23 separate Bloom filters. Each Bloom filter scans for a specific signature length. Upon a signature match, the analyzer is queried to determine if a true match has occurred. If a true match is found, the ID number of the corresponding rule is sent out of the scanner.

Bloom Filter Configuration

A Bloom filter allows for quickly determining membership in a large database [21]. The byte window is queried by hashing all the bits in the window and checking if hashed locations are set to '1'. The equation for the false positive of each Bloom filter engine is governed by Equation 3.1 [37]. The value is plotted as n, the number of strings, varies in Figure 3.3.

$$f = (1 - (1 - \frac{1}{16384})^{8n})^8 \tag{3.1}$$

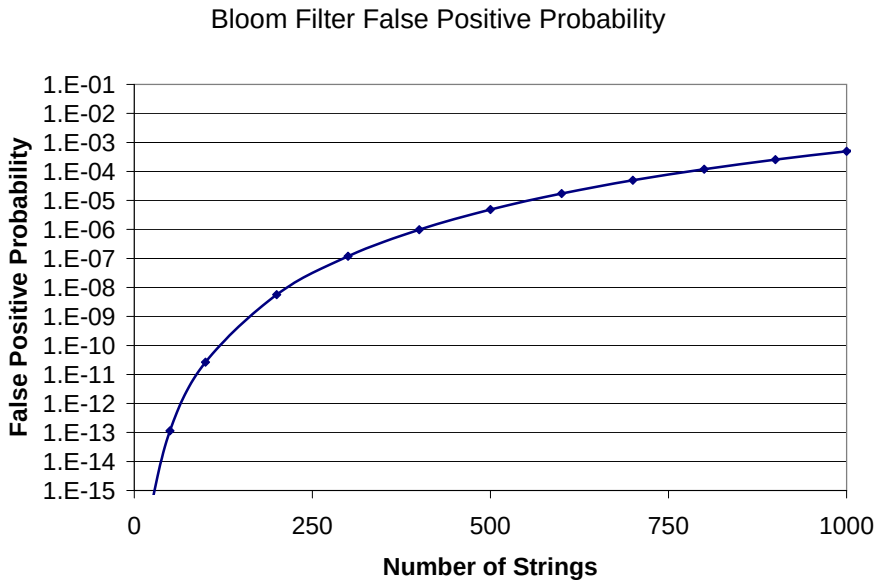

Figure 3.3: False positive probability versus the number of stored strings.

Each Bloom filter engine has a negligible false positive probability when less than 800 strings are stored. The number of signatures the system can hold at this low false positive probability is $23 * 800 = 18,400$. The most Snort signatures associated with any particular signature length is 163, which gives a false positive probability of 3E-8.

A Bloom filter, as shown in Figure 3.4, is composed of four partial Bloom filters (PBFs), logic to perform the eight separate hash calculations, and a control unit. To efficiently map the Bloom filter into FPGA logic, as described in [36], several PBFs are instantiated. A PBF is a wrapper around a block RAM. On the Xilinx Virtex XCV2000E, there are 160 4096-bit block RAMs. Each block RAM provides one clock cycle read and write latency.

The hash function used is a function well-tuned for implementation in hardware [92]. It consists of using a matrix of *AND* and *XOR* gates. Eight hash functions are computed, and the i^{th} hash function is computed over the l input bits of $B = \{b_1, b_2, b_3, \cdots b_l\}$ from a constant random value seed $X_i = \{x_{i,1}, x_{i,2}, x_{i,3}, \cdots x_{i,l}\}$ by using Equation 3.2. The elements of X_i are the length of the hash to be computed.

$$h_i(X_i) = b_1 \cdot x_{i,1} \oplus b_2 \cdot x_{i,2} \oplus b_3 \cdot x_{i,3} \oplus \cdots \oplus b_l \cdot x_{i,l} \tag{3.2}$$

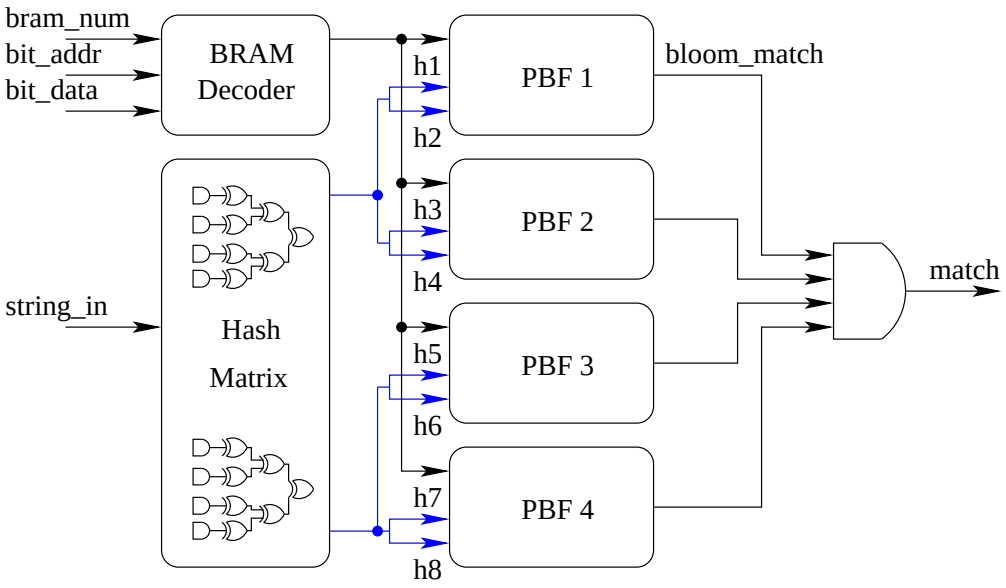

Figure 3.4: A Bloom filter instantiates a BRAM decoder, a hash matrix, and four partial Bloom filters. By using four PBFs, a 16,384-bit vector is created. When each PBF signals a match, the input string exists in the database with a certain probability.

Consider an input signature having five bits ($l = 5$) and a hash length of three bits. If $B = 01011$ and $X = \{011, 100, 001, 011, 110\}$, the resulting hash is $100 \oplus 011 \oplus 110 = 001$.

To control a Bloom filter, three signals are used: *bram_num*, *bit_addr*, and *bit_data*. The *bram_num* determines which PBF block RAM to select. The other two signals are used to set/reset the appropriate bit in the bit-vector.

The hash matrix converts the incoming string into eight 12-bit addresses to be queried in PBFs. The Bloom filter is pipelined to take two cycles to compute the hashes and three to determine if the signature exists in the database.

A PBF, as shown in Figure 3.5, uses two hash functions to index 4,096 bits of the bit-vector. The PBF allows back-to-back queries to be performed using the 3-stage pipeline shown. In the first stage, the inputs are registered. In the second stage, hash addresses are queried. In the final stage, the results are latched and combined to signal whether the signature matched or not.

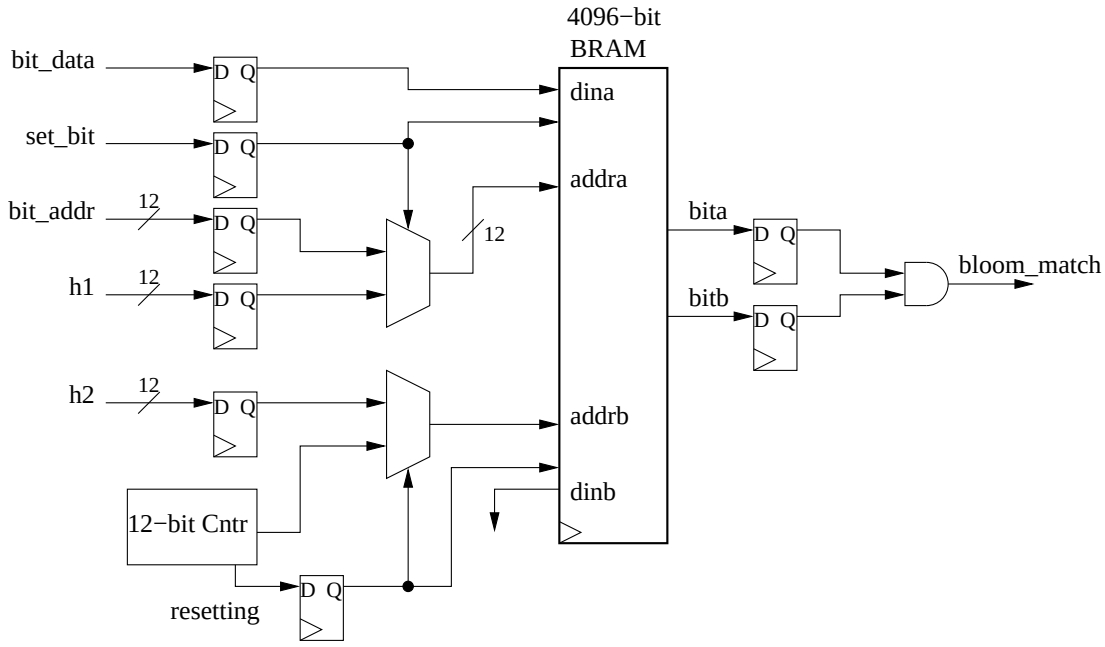

Figure 3.5: A partial Bloom filter (PBF) is a 3-stage pipeline allowing back-to-back queries of membership for different signatures. Two hashes index into the vector that is implemented in dual-port block RAM.

Hash Table Analyzer

The hash table analyzer, as shown in Figure 3.6, stores in SDRAM the signatures that have been programmed in Bloom filters. The same type of hash function is used as was used for the Bloom filters, except the hash length is 20 bits instead of 12.

When a Bloom filter signals a match, the matching signature is passed to the hash table analyzer. A hash is calculated over the incoming signature, and this address is looked up in SDRAM. If the string held at the memory location matches the input signature, *true match* is asserted.

If the string in memory does not match, there are two potential reasons. First, this was a false positive, and the string does not exist in the database. Second, two signatures happened to map to the same memory location resulting in a collision. To handle collisions, probing is performed by adding constants to the hash address. If there is a mismatch, the constant is changed from 0 to 1 to search the next address in SDRAM. The constant address is modified in a quadratic fashion (0, 1, 2, 4, 8, 16...). The use of an occupied and deleted bit was added to the SDRAM entry to ease dealing with hash-collision issues.

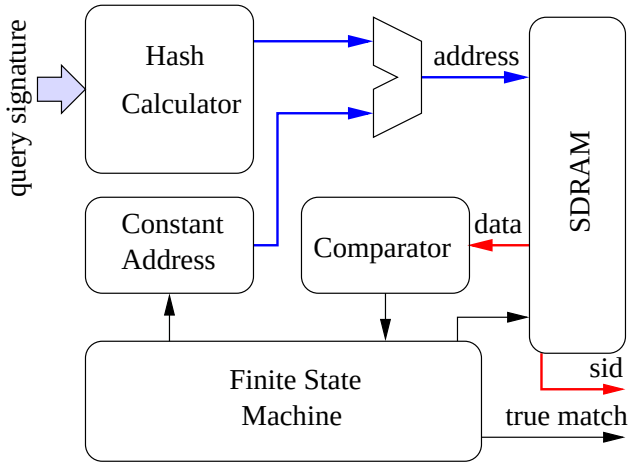

Figure 3.6: The hash table consists of a hash calculator, a finite state machine, and a comparator. The hash table resolves collisions by performing quadratic probing.

This hash technique was found to result in few collisions. There are 2^{20} memory slots available to hold signature records. With 20,000 signatures loaded into the analyzer, 220 first level collisions and four second level collisions were witnessed. With the 2,107 signatures from the Snort database, two collisions were simulated.

The layout of a hash record is shown in Figure 3.7. To ensure that the hash chain will be followed in the case of a collision, the FSM checks the deleted bit to determine whether to increase the probe address. The occupied bit specifies whether the given location is valid. If there is no record, the FSM stops the search. The hash record holds up to 32 byte strings. To simplify the design, signatures that are not 32 bytes are extended to 32 bytes by adding zeroes in the upper bytes. Finally, the string ID (SID) is the identification number returned from the analyzer. This ID is sent to the header processor.

Two implementations of this circuit exist, one that uses SRAM and one that uses SDRAM. The SRAM implementation has approximately the same latency as the SDRAM implementation, but it stores fewer signatures. The SRAM implementation transfers the data from memory in twice as many clock cycles as the SDRAM implementation because only 32 bits can be read from SRAM per clock cycle, while 64 bits can be read from SDRAM per clock cycle.

Assuming only one hash record has to be retrieved, the latency from a Bloom filter detecting a match to resolving whether the match is true is 20 clock cycles. During this time, the pipeline is paused so as not to be overrun with additional queries.

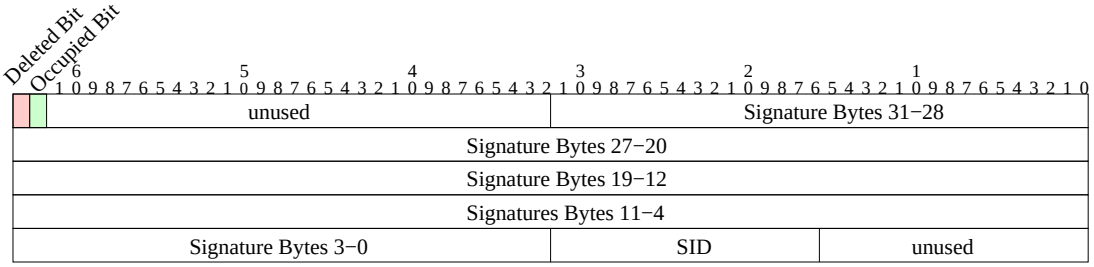

Figure 3.7: The hash record allocates up to 32 bytes for the signature, two bytes for the string ID, and holds two bits of information about the state of the current entry.

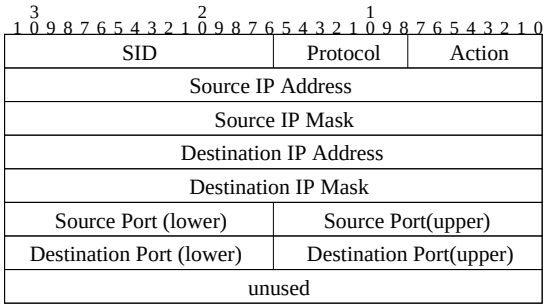

Figure 3.8: The header record allocates 32 bytes to hold the source IP address, source IP mask, destination IP address, destination IP mask, source port range, destination port range, protocol, and action to take.

3.4.3 Header Processor

The header processor implements the comparison circuitry that would be found in a TCAM. When a content match occurs, the analyzer returns the corresponding signature ID that is used to index SRAM to retrieve the header entry of Figure 3.8. SRAM stores the protocol, the source and destination IP addresses, masks, source and destination ports, and the action to take. The latency of retrieving a record and determining if the header matches is 15 clock cycles. The available actions are to alert the software controller a match occurred, to drop the offending packet, to return the offending packet to software for further analysis, or to return the offending packet to software while dropping it from the network.

To determine if an IP address matches the header rule, the address is XOR'd with the stored IP address. The result of this is NAND'd with the mask. If the result is a vector of 32 ones, the IP address matches. To determine if a port matches, it is checked to see if it is greater than or equal to a lower bound and less than or equal to

an upper bound. If both of these conditions are satisfied, the port matches. A direct comparison is done on the protocol field.

String matching is performed before header processing, which is the opposite of the approach used in Snort. There were several reasons for this decision. First, since a signature appears in so few rules in general, the presence of the signature can efficiently trigger the header classification. Since over 95% of rules in Snort specify headers of the form *EXTERNAL_NET to HOME_NET*, a header rule will almost always match, and no processing reduction occurs. Second, many headers can match simultaneously. This complicates the correlation between matching headers and signatures. Use of the signature detection as a pre-filter reduces the number of headers to check from hundreds to only a handful.

3.4.4 Control Packet Processor

The control packet processor is the main finite state machine (FSM) of the system. This FSM keeps track of what part of the IP packet is currently entering the system. In addition, this component programs the various components when control packets are sent to the system. Control packets are distinguished from other packets by listening on a specific programming port. The following operation codes are supported and are shown in detail in Appendix A:

- Add/remove string to/from the hash table analyzer
- Program bit(s) in a Bloom filter
- Add/remove a header table entry to/from SRAM
- Change the destination of alert messages
- Read a header table entry
- Read a statistic
- Test whether circuit is operational
- Clear all rules from the system

3.4.5 Statistics Module

The statistics module allows for 256 different events to be stored in block RAMs [15]. Snort Lite was configured to count the number of incoming IP, UDP, and TCP packets, the number of analyzer queries, the number of true matches, the number of matching headers, the number of control packets received (sorted by opcode), and the number of times each of the four actions was taken.

3.4.6 Alert Generator

The alert generator generates UDP packets to send information to a software process. Rule matches trigger the generation of alert packets destined for software, which track how many times the rule matched. These alert packets can also be sent to a program that graphically displays matches per second, as shown in Figure A.17. Additionally, the alert generator can bundle up packets that matched rules and send them to software for further inspection.

3.4.7 Hardware Infrastructure

Layered Protocol Wrappers

The layered protocol wrappers provide an interface that identifies the fields within an IP packet [23, 77]. This eases the design of an IP-based networking application by allowing it to operate at OSI layer three.

Memory Controllers

The SRAM controller provides two arbitrated interfaces for access to a 2 MB ZBT SRAM [119]. A request and grant protocol is used to access SRAM.

The SDRAM controller provides three arbitrated interfaces to SDRAM [39]. One is for reading only, one is for writing only, and one is for reading and writing. The read/write interface is used to access a bank of 64 MB SDRAM. The SDRAM controller also provides a simple request/grant interface with burst transfers.

Buffers

Buffers are used throughout the system to store IP packets before processing. There are two primary buffers: the input buffer and the output buffer. The input buffer

Table 3.1: A summary of Snort Lite resources used by component in a Xilinx Virtex XCV2000E-8 FPGA.

Component	Function Generators	Flip Flops	Block RAMs
Bloom Filters	14082	8802	92
CPP	148	317	0
Hash Table	1436	1475	0
Header Processor	386	594	3
Input Buffer	165	114	10
Output Buffer and Alert Generator	962	727	24
Protocol Wrappers	2502	3103	27
SDRAM Controller	516	414	0
SRAM Controller	66	137	0
Statistics Module	91	146	2
Miscellaneous	174	462	0
Design Totals	20528	16291	158

converts a 4-byte word stream into a byte stream that passes by the Bloom engines. The output buffer reassembles the byte stream into 4-byte words and outputs the packet.

3.5 Implementation Results

Snort Lite used 55% of the 4-input look-up tables (LUTs), 77% of the logic slices, and 99% of the block RAMs on a Virtex XCV2000E-8 FPGA. The number of function generators, flip-flops, and block RAMs used by the major components is itemized in Table 3.1. The design synthesized at 62.802 MHz.

FPGA features were heavily exploited in this design. The limiting resource on the FPGA turned out to be the distributed block RAM. Note that Bloom filters required the most resources and the protocol wrappers required the second most.

3.6 Memory Bandwidth Requirements

In order to determine if a Bloom filter match was a true positive or a false positive, SDRAM was read. Theoretically, each of the 23 Bloom engines could trigger a match during each clock cycle. This would result in the need to retrieve 920 bytes from

SDRAM per clock cycle, which requires 460 Gbps memory bandwidth. For this reason, the pipeline was paused when matches occurred.

3.7 Observations

3.7.1 Security & Adaptability

A randomized algorithm was used to hash strings. One might think that an attacker could send packets that contain strings that hash to known offending signatures, which in turn would generate many false positives. It is possible to guard against such a scenario by changing the random values used to generate each of the functions used by the hashing circuits. New hash functions can then be dynamically configured into the FPGA logic.

The use of UDP packets to control the system was a security concern. The system listened for packets from a specific host. A secure control and configuration scheme was explored to solve the spoofing problem in [109]. Control messages from the real controller were encrypted using AES and decrypted in reconfigurable hardware.

In an evolving network, a circuit designer does not know what threats will appear in the future. For this reason, the design of the system was made to be modular, and it can be changed by reconfiguration [49]. For instance, suppose the need arises to be able to process rules that contain signatures that are longer than 24 bytes and truncation is not acceptable. The design is configurable to allow expansion to scan for 32 byte signatures while removing scans of smaller length signatures. Only minor changes to the design flow are needed to expand to greater lengths.

The network characteristics may change also. The bandwidth of this design can be scaled beyond 2 Gbps by instantiating parallel copies of the Bloom filters to scan multiple bytes per clock cycle. There are, however, trade-offs in increasing parallelism. Because this circuit already fully utilizes the resources of the FPGA, using parallel copies of the scanning technology limits the number of signature lengths that can be scanned. While 23 separate lengths are supported at 500 Mbps, only six are supported at 2 Gbps.

3.7.2 Additions & Expansions

The requirement for one string per header rule could be relaxed by generalizing the hash table to store more rule IDs in a hash record. Currently, up to 14 additional

SIDs could be returned without major modification. The SDRAM controller operates on bursts that are powers of two, and the hash table already retrieves eight 64-bit words, three of which are unused. The Snort database does not associate a string with more than 15 headers in all but 13 cases.

To support the 20% of rules that require more than one signature match for the rule to match, additional logic would be required in the header processor. Counters about how many of the signatures have been found would be necessary. Software currently performs this task.

Additional processing of other header fields, such as the time-to-live, flags, and packet sequence number could be performed. This task is not technically difficult to perform. Storage of additional fields in the header record would be required to support additional features.

3.7.3 Lessons Learned

As this design was intended to be a baseline circuit for future applications, many important points were gleaned. The use of a Bloom engine for each length required approximately 60% of the total block RAMs, which limits the amount of on-chip buffering available.

Second, the buffering requirements of the system are critical to operation. Ideally, the system should never back pressure upstream modules. The conversion from a 4-byte stream to a 1-byte stream forces these buffers to be large. Snort Lite can store three full-size packets using block RAMs. To buffer more, external memory is necessary.

Third, the byte advance of the processing pipeline is unacceptable for future applications. It severely cripples the potential throughput of the system. Through the use of parallel processing pipelines, the full line rate can be achieved at the sacrifice of the quantity of Bloom engines that can be used.

Finally, the timing of the system is hindered almost exclusively by the fan-out of the pipeline. The lower order bytes in the windows being monitored go to each Bloom engine where the hash is calculated. Future designs can explore shifting the windows for each Bloom engine about the pipeline to reduce the fan-out.

Chapter 4

Snort Intrusion Filter for TCP

The Snort Intrusion Filter for TCP (SIFT) operates as a preprocessor to filter out traffic destined for a passive intrusion monitor PC running Snort. The system drops IP packets destined to a sensor that do not contain keywords or questionable headers. Packets containing questionable criteria are forwarded, unaltered, to a PC running Snort. Most network traffic does not need to be inspected by an intrusion system. SIFT was devised to limit the workload of passive software intrusion monitors.

By maintaining statistics about suspect packets, data about how many packets match rules was obtained. This metric aids designers in optimizing rule processing architectures. SIFT was implemented and tested with live Internet traffic from a campus Internet tap.

4.1 Design Objectives

The main goal of SIFT was to perform intrusion detection analysis on real network traffic. The testing results revealed how often signatures and headers matched. This information provided crucial insight about how to optimize rule processing systems to process real traffic. Results from testing also confirmed that software-based PCs running intrusion detection software could not keep up with the traffic found in actual networks [61, 94].

To process traffic at a high throughput, SIFT utilized multiple parallel Bloom engines. The mechanism to send commands to program rules into SIFT was built on top of a generic communication wrapper.

SIFT utilized TCP stream reconstruction to scan for signatures that can cross packet boundaries. In order to track data in packets that can be interleaved in time, SIFT used memory to store and retrieve context information for each traffic flow.

SIFT acted as a passive monitor to filter out harmless traffic. In order to implement intrusion detection functions, SIFT operated in real-time, was highly adaptable, and succinctly reported status.

4.2 Design Decisions

4.2.1 Characteristics

Intrusion rule databases continue to expand as new threats emerge. However, as the number of header rules and signatures to match increases, the CPU on a PC running Snort becomes fully utilized, and the number of packets dropped by the PC increases. To be an effective monitor, the intrusion system must process all packets.

The motivation for this architecture was to off-load Snort rule processing from a PC to hardware. After analyzing the current Snort rule database, a method was found to separate harmful traffic from safe traffic. There are 292 unique header rules, and 168 of these do not have a signature associated with them. Thus, for all but 168 of the 2,464 rules, signature detection can be used as filter criteria. If one of the 2,107 signatures is found, then that is sufficient to forward the packet to software for further examination. There are 233 rules that contain regular expressions. However, every rule that contains regular expressions also contains separate static signatures. Thus, an implementation of regular expression scanning to flag harmful packets was not necessary. In this implementation of the system, the static signatures found in rules were used as search criteria.

4.2.2 Architectural Needs

Bloom filters were used to support the large number of strings found in Snort rules. To increase the throughput, four parallel Bloom engines per length were used.

To support operating on TCP flows, an efficient context storage engine was necessary. A context storage engine was implemented for this circuit that accessed SDRAM to store the current state of the byte pipeline when a different flow arrived and retrieved the old flow context.

An efficient control mechanism was necessary to relay information to/from a control PC and the hardware device. The communication wrapper was developed for this purpose.

To support the header-only rules, a hardware module was developed to process packet headers in parallel.

Finally, to display system statistics, a SNMP agent was developed to periodically report system statistics to a software process, where they are plotted using MRTG.

4.3 Features

The SIFT architecture introduced a number of features:

- Support for case-insensitive string matching across TCP packet boundaries
- Support for up to 5,000 signatures
- Ability to scan payloads for five different signature lengths between 2 and 32 bytes
- Ability to reconfigure hardware to change the amount of resources used to scan for different signature lengths
- Support for fully custom header rules to be processed in parallel
- Support to forward the content of an offending packet to another machine for further inspection
- Maximum throughput of 2.5 Gbps on the FPX platform

4.4 Context Switching

A very important aspect of the design of SIFT was how context switching was performed. To scan TCP flows, information about the state of the flow must be maintained to reassemble the stream of data. In this circuit, the last 32 bytes of data for each flow was saved. When a TCP packet from a different flow entered the system, the last 32 bytes of that flow were retrieved from SDRAM so that a byte pipeline can be primed for the new TCP data. At the end of the TCP packet the last 32 bytes of data were sent to SDRAM for storage.

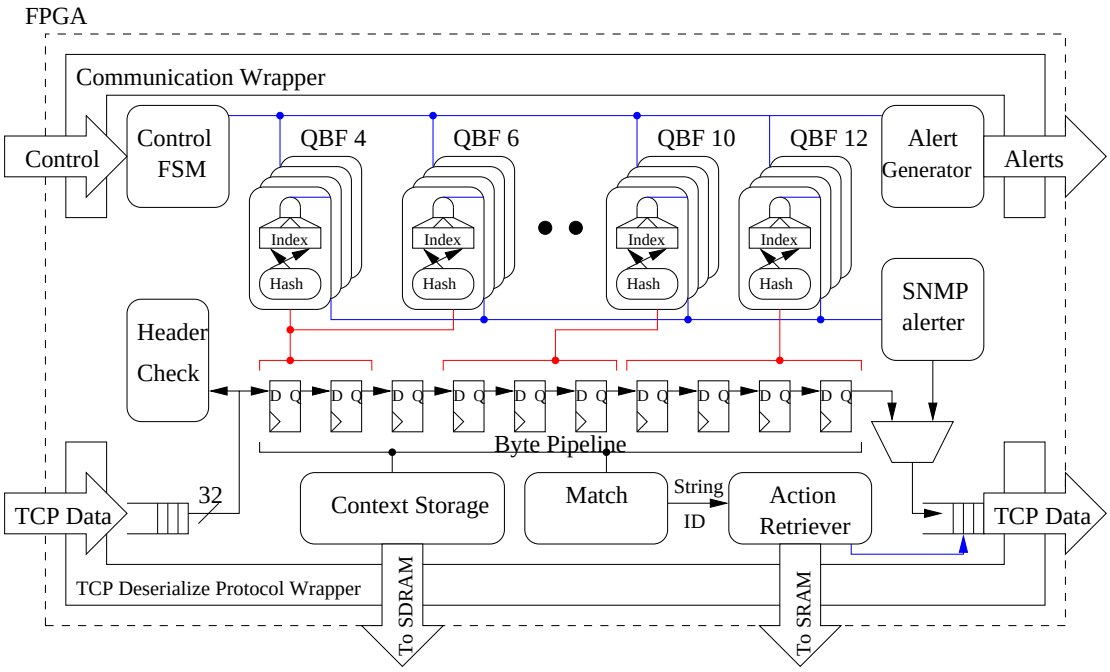

Figure 4.1: SIFT used Bloom filters for string matching and custom logic for header classification. External memory was used to store flow context. System statistics were periodically reported out of the system to a control PC.

4.5 Architecture

4.5.1 Overview

A block diagram of SIFT is shown in Figure 4.1. IP data packets enter the system via the TCP deserialize wrapper [95, 99], where they are annotated with control signals to mark specific locations in the packet, such as the beginning and ending of IP packet headers. This data is buffered and sent to the header check component to determine if one of the 168 header-only rules match the current packet. The packet is sent to the content pipeline where each byte-offset is searched for signatures by Bloom filters. If a match is found, the matching engine reports to the match decoder to determine the signature ID. This ID is sent to the action retrieval unit to determine the action to take on the packet. The default case forwards the packet. If no signature is found, the packet is filtered. If a signature is found or a header-only rule matches, then the packet is sent out of the system to a Snort PC via the outbound side of the TCP deserialize wrapper.

The system is incrementally programmable. Signatures can be added or deleted quickly via User Datagram Protocol (UDP) control packets that are sent directly

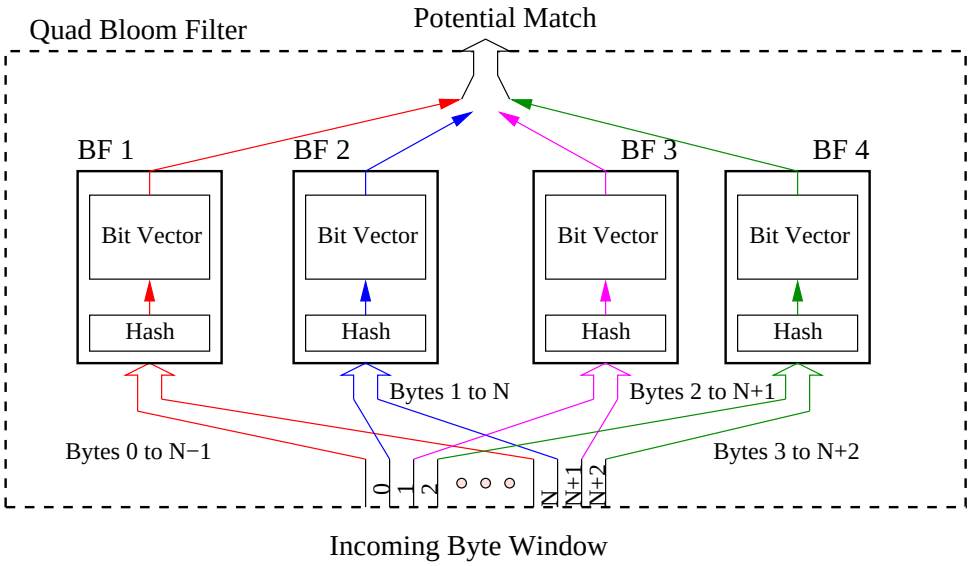

Figure 4.2: A quad Bloom filter instantiates four Bloom filters engines. Each Bloom filter is passed a byte-offset window in order to perform a query.

to the communication wrapper interface from a software system that uses counting Bloom filters to determine when to set/reset bits within the indices. These control packets are sent to the control FSM, where they are forwarded to the appropriate engine for integration. Real-time statistics are sent out of the system via the SNMP alerter.

To accommodate string matching across TCP packets, a context storage device holds the last 32 bytes of the current flow going through the system. When a different flow enters the system, the new flow's data is loaded and the current flow's data is stored.

4.5.2 Quad Bloom Filters

In order to scan for strings that appear in the input stream at each of the four possible byte offsets, it is necessary to process four different strings in each clock cycle. A quad Bloom filter (QBF), as shown in Figure 4.2, instantiates four Bloom filters for this purpose. An input signature three bytes longer than the signature width is passed to the QBF in order to scan each byte-offset version of the byte pipeline. By using a QBF, SIFT can process four bytes of data per clock cycle.

To understand the progression of bytes through the pipeline, consider Figure 4.3. Suppose the current value of the pipeline is ABCDEFGH, as in Figure 4.3a.

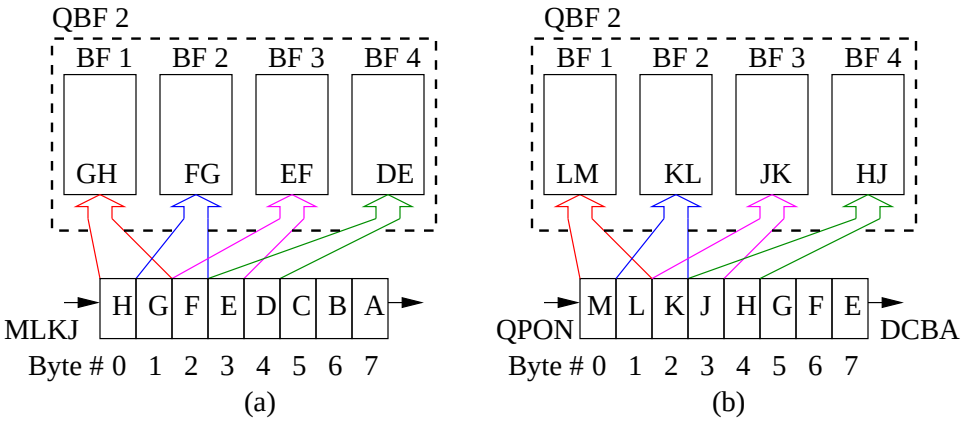

Figure 4.3: An example of bytes moving through the pipeline.

A QBF that scans for two byte strings receives the input DEFGH. GH is sent to BF 1, FG is sent to BF 2, EF is sent to BF 3, and DE is sent to BF 4. The byte pipeline advances by four bytes, as shown in Figure 4.3b, to EFGHIJKLM.

4.5.3 Header Check

The header check component first separates the header-only rules into logic blocks based on protocol: TCP, UDP, ICMP, and other IP. Most of the 168 Snort header-only rules look for specific TCP/UDP port numbers or protocol-specific features like a sequence number. Each of the 168 header-only rules are checked in parallel, and a rule match is declared if any of the headers match.

4.5.4 Action Retriever

An action retriever interacts with SRAM to determine the programmable action to perform when a signature matches. Each signature has a 16-bit ID number. This component uses the signature ID to look up the action to take. Available actions are to generate an alert, to filter (drop) the packet, or to return the data in the packet to the monitoring host.

4.5.5 SNMP Alerter

Hardware and software tools were assembled to display live traffic statistics. The SNMP (Simple Network Management Protocol) alerter periodically reports system-wide statistics to a software-based SNMP agent. The SNMP agent is queried by

MRTG (multi router traffic grapher) to graphically display statistics. SIFT tracked the following 18 events: incoming TCP packets, incoming UDP packets, incoming ICMP packets, incoming IP packets (non TCP, UDP, ICMP), outgoing TCP packets, outgoing UDP packets, outgoing ICMP packets, outgoing IP packets (non TCP, UDP, ICMP), number of matching headers, string matches for QBF engine 1, string matches for QBF engine 2, string matches for QBF engine 3, string matches for QBF engine 4, string matches for QBF engine 5, number of alert actions, number of filter actions, number of return actions, and number of filter and return actions.

4.5.6 Alert Generator

The alert generator bundles and transmits alert messages or entire packets to a software controller where the data can be logged and further inspection can be performed. The alert generator formats output for use with the communication wrapper.

4.5.7 Communication Wrapper

The communication wrapper was devised as a way to simplify the communication between hardware processing modules and software by abstracting the underlying transport mechanism and providing reliable communication channel. The interface between the communication wrapper and a network data processing module is shown in Figure 4.4.

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

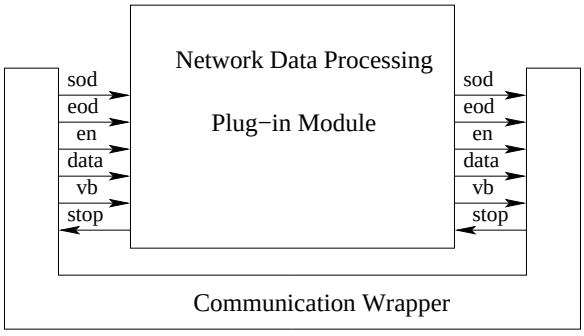

Figure 4.4: The communication wrapper transports control information between modules and software. The wrapper abstracts the underlying transmission requirements needed for communication, be it across devices or across the FPGA.

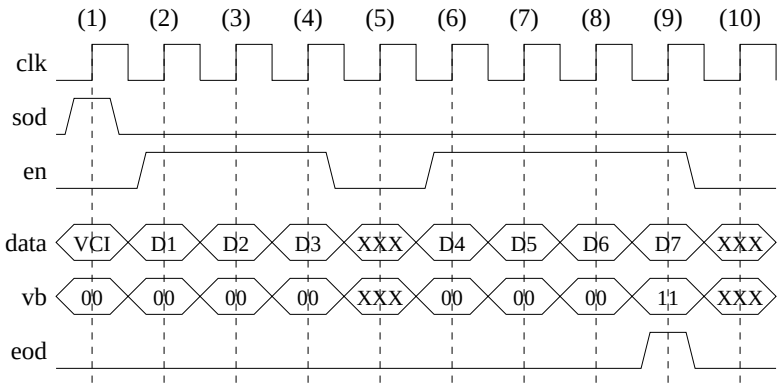

Figure 4.5: An example timing diagram of the communication wrapper interface.

flow itself; the number of valid bytes (*vb*) on the *data* signal; a busy signal (*stop*) used to temporarily halt the flow of data in the case of a backlog.

An example timing diagram is given in Figure 4.5. Data transmission begins by the assertion of *sod*. In this clock cycle, the data bus holds the VCI word, which is used by applications to determine how to process data. In the next cycle, *en* is asserted, meaning valid data is on the data bus. At the same time, the *vb* signal specifies how many of the bytes on the bus are valid. If the value is "01", one byte is valid; if "10", two bytes are valid; if "11", three bytes are valid; if "00", all four bytes are valid. In clock cycle five, *en* is de-asserted, indicating that the value on the data bus is not valid. The *en* signal is reasserted in clock cycle six, and the data on the data bus is valid. The data stream ends in clock cycle nine, which is indicated by the assertion of *eod*. In this example, the last word contains three valid bytes.

4.5.8 TCP Lite Wrapper

The TCP Lite wrapper was developed for use with the TCP Processing circuit developed in [96]. The TCP processor provides in-order delivery of TCP streams to processing applications. The wrapper decodes the TCP stream traffic, providing interface signals to identify the beginning and end of IP packet headers and where TCP data begins and ends in the data stream.

In addition to extracting an in-order stream of data from each TCP/IP flow that passes through the system, a flow identification value for each TCP stream is given. This identifier is used to store and retrieve context information from an SDRAM buffer for each flow.

4.5.9 SDRAM Buffer

A context storage component interfaces with one bank of SDRAM to retrieve and store the byte pipeline state. When a new packet arrives, the flow ID associated with the packet is used to retrieve flow context. The last 32 bytes of TCP data is held in SDRAM. Flow context is fetched while the previous packet is being processed, preventing the pipeline from stalling. Using a 64 MB SDRAM, context for over two million simultaneous TCP flows can be tracked.

Retrieving 32 bytes of data from SDRAM requires 20 clock cycles. The new ID's data is retrieved before storing the old ID's data.

4.5.10 Input/Output Buffering

An entire packet is buffered before streaming it through the pipeline. This simplifies the content scanning since the pipeline never has to be paused. The input buffer can hold five 1500-byte packets.

Packets are buffered on the output side as well as on the input side in order to wait for the action to take to be resolved. The output buffer is capable of holding three 1500-byte packets.

4.6 Implementation Results

The implementation of SIFT in a Xilinx Virtex XCV2000E-8 FPGA utilized 84% of the logic slices, 58% of the look-up tables, and 96% of the block RAMs. Details of the number of function generators, flip-flops, and block RAMs used by each component are given in Table 4.1. The design synthesized to 80 MHz, providing a throughput of 2.5 Gbps.

4.7 Memory Bandwidth Requirements

The architecture was optimized to reduce pipeline stalls. Pipeline stalls could happen for a number of reasons. First, the retrieval of state information could pause the pipeline when a flood of small payload packets belonging to different flows arrive back-to-back. The latency to memory is greater than the time it takes to receive a small packet. A minimum size TCP packet requires 14 clock cycles to enter the system, and the memory access time is 20 clock cycles. Second, a barrage of string

Table 4.1: A summary of resources used by SIFT components in a Xilinx Virtex XCV2000E-8 FPGA.

Component	Function Generators	Flip Flops	Block RAMs
Quad Bloom Filters	10201	7681	80
Control FSM	15	5	0
Header Check	276	268	0
Buffers	327	409	31
Alert Generator	218	226	3
Communication Wrapper	912	1103	19
TCP Lite Wrapper	2060	1542	16
SDRAM Buffer	792	936	0
Match Decoder	15	5	0
Action Retriever	1435	1621	1
SNMP Alerter	848	906	0
Miscellaneous	3188	4180	4
Design Totals	20287	18882	154

matches could back-log the action retrieval unit. This only occurs if a buffer holding matching signatures has more than eight outstanding matches.

The architecture was designed such that the SDRAM latency was masked by the progression of previous packets through the pipeline. The instant a packet arrived, the SDRAM request was sent. The SDRAM access was performed while the packet was buffered.

Stalls caused by a full match buffer are more difficult to combat. The match buffer implemented in this design holds 16 signatures. The pipeline stalls when the number of signatures buffered exceeds this limit. There are two ways to reduce this stall. The first is to make the buffer larger, allowing more outstanding signatures to wait for identification. The second solution is to use both available banks of SRAM to perform action retrieval. In this way, two signatures could be processed in the time it takes to resolve one.

4.8 Observations

4.8.1 String Matching

String matching performed with Bloom filters can generate false positives. In an earlier version of the design, a false positive resolving circuit verified that a string was present. However, the false positive resolver was removed since occasional outputs of safe packets was acceptable because they were post-processed in software.

By using four parallel Bloom engines to scan for a specific length signature, the amount of resources left in the FPGA to scan for different length strings was limited. To reduce the number of different length strings search, SIFT truncated some signatures. Truncation created another type of false positive. That is, a truncated signature may generate an alert, whereas the original signature was not present. This is discussed in more depth in Chapter 6.

Chapter 5

Rule Processing Framework

The Rule Processing Framework (RPF) was designed to allow complete generality of intrusion rules by interpreting rule components as logical equations. Header processing and string matching are performed in separate components, each of which inform the rule processor what criteria matched, and the rule processor determines whether rules match. The RPF operates on TCP flows.

In this chapter, the generic framework for implementing a rule processing system in reconfigurable hardware is discussed. 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 the system in order to extend the functionality to support the remaining set of Snort rules in modular components. Reconfigurability and flexibility are key components of the system that enable it to adapt to protect Internet systems from threats including malicious worms, computer viruses, and network intruders.

The framework can process 32,768 complex rules at data rates of 2.5 Gbps on the FPX platform. Systems to handle data at 10 Gbps rates can be built today using the same framework in the most recent reconfigurable hardware devices.

5.1 Design Objectives

To accommodate features necessary for complex rule processing systems, including support for regular expression scanning, fixed string scanning, header processing, and multiple criteria per rule, a generic rule processing architecture was developed. The RPF combines components to support these features into a single rule processing structure.

In order to adapt to new threats, the ability to add a processing module was valuable. A generic way to communicate between processing modules and the rule processor was developed. The communication wrapper allows modules to be added.

TCP stream processing is a key aspect of content processing. Stream processing allows data spanning packet boundaries to be processed as if there were no separation in time between the arrival of data. However, since the fundamental unit of any NIDS is packets, a way to store context for millions of flows was needed.

The RPF had several desirable features: it operated in real-time, it was adaptable, and it operated efficiently. Rule updates were written to the system via control messages that updated internal data structures. Intrusion information was sent directly from hardware to a control host for logging and display.

The key aspect of the RPF is flexibility. The nature of future attacks to the Internet's infrastructure is difficult to predict. The ability to protect the Internet from unknown types of future attacks requires that the rule processing system adapt. The use of reconfigurable hardware in the system supports future enhancements.

5.2 Design Decisions

5.2.1 System Layout

A large amount of hardware resources are required to implement a complete rule processing system. The RPF allows the main components of rule processing to be separate. The rule processor could be in a different area of the FPGA or it could be implemented on a separate device. The RPF that was developed implemented the components of rule processing on different FPGAs.

Components communicated matching criteria found in a given flow (or packet) using the format of Figure 5.1. The header consists of flags and a flow ID. The flags specify whether the following ID values are header or string IDs. The flow ID specifies to what TCP flow these matches belong. Subsequent words contain identification values of matching criteria.

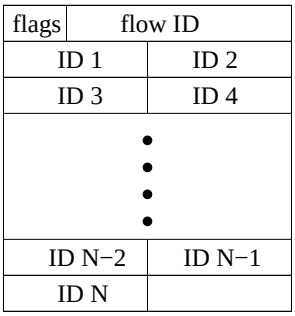

Figure 5.1: The data format for communicating matching rules and signatures consists of flags, a flow ID field, and a list of matching ID numbers.

5.3 Features

The RPF has the following features:

- Expandable support for all features found in Snort (dependent on processing modules capabilities)
- Support for a total of 32,768 rules
- Support for up to 32,768 unique header rules
- Support for up to 32,768 unique signatures
- Support for rules to have up to 15 embedded signatures or regular expressions
- Support for 1 header rule per rule (but expandable to be up to 15)
- Ability to add/remove modular processing components
- Support for incremental rule updates via control messages
- Support to operate on TCP flows as well as IP packets
- Maximum throughput of 2.5 Gbps on the FPX
- Succinct or verbose reporting options

The novel aspect of this framework is that multiple techniques can be used simultaneously to perform rule processing in hardware. To the rule processor, header rules and signatures are ID numbers reported by custom processing modules. For example, a processing module that efficiently scans for regular expressions can be used

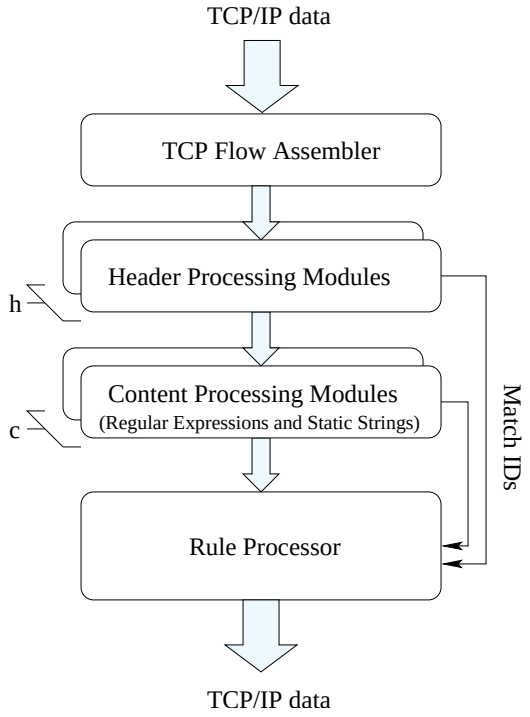

Figure 5.2: The rule-processing framework allows for header processing and content scanning components to be added or removed without affecting rule processing. A standard communication mechanism was adopted to allow for integration of multiple components.

in tandem with a processing module that has been optimized for static signatures. In this way, the rule processor supports all features currently found in Snort by allowing modular components to be integrated into the system that have been optimized for each particular need.

5.4 System Overview

The RPF utilizes processing modules to determine which header rules and signatures have been found. The processing modules send information to the rule processor, where correlation between header and signature matches determines whether rules match.

Data flows through the system from a TCP flow assembler to h header processors and c content scanners, as shown in Figure 5.2. The flow assembler provides ordered TCP data to content scanners. 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.

To the rule processor, rules have the form:

$$< action > \ H_{ID} \ \wedge \ (S1_{ID} \ \wedge \ S2_{ID} \ \wedge \ \cdots \ \wedge \ Sn_{ID});$$

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 through control packets from a management console.

5.5 Architecture

The rule processor is the key component of the RPF, and it is shown in Figure 5.3. To achieve high performance, the rule processor was pipelined with seven pipeline stages. All communication was performed using the communication wrappers. The rule processor handles two forms of inputs and a single form of output. Control information enters on a control interface. Matching header and signature IDs enter on an ID interface. Rule match information and actions to take are output the alert interface.

5.5.1 Pipeline Stages

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, the IDs 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) are split in this stage. CIDs advance to the second stage of the pipeline, while HIDs are forwarded to stage 5.

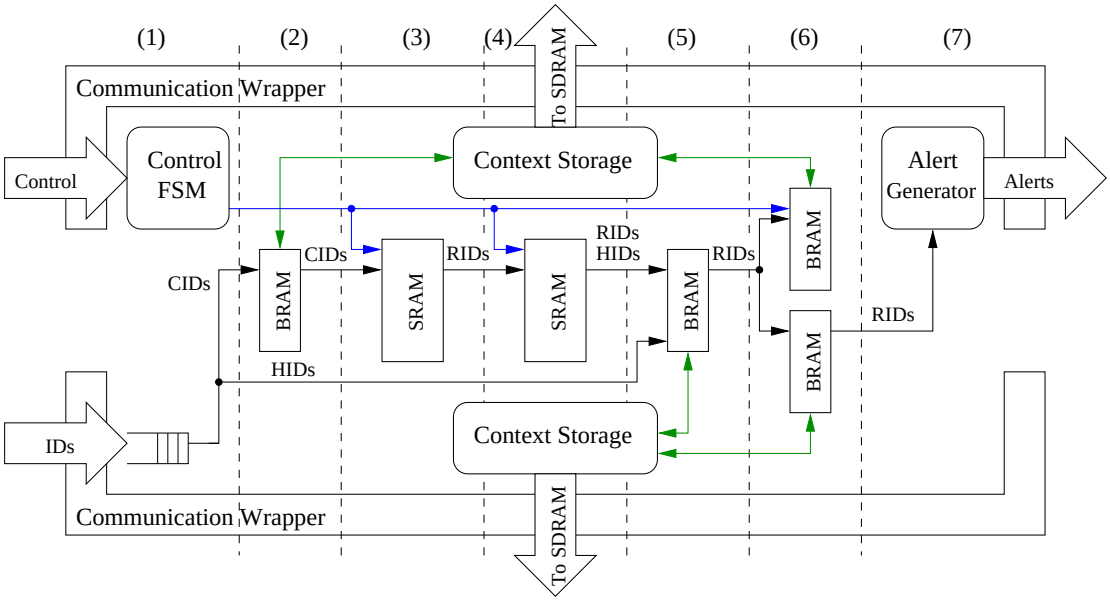

Figure 5.3: The rule processor consists of seven stages: (1) input, (2) content ID check, (3) rule ID retrieval, (4) header ID mapping, (5) header ID check, (6) count check, and (7) alert output.

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 indexed by the CID. If the indexed bucket is set, no further processing is performed because the signature was already detected, but a rule has not matched yet. If the indexed bucket 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.

Stage 3: Rule ID Retrieval

Stage 3 receives uniquely occurring matching signatures, and a reverse index look-up [127] is performed to determine what rule IDs (RIDs) are associated with the signature. 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, four RIDs are returned since *".mp3"*

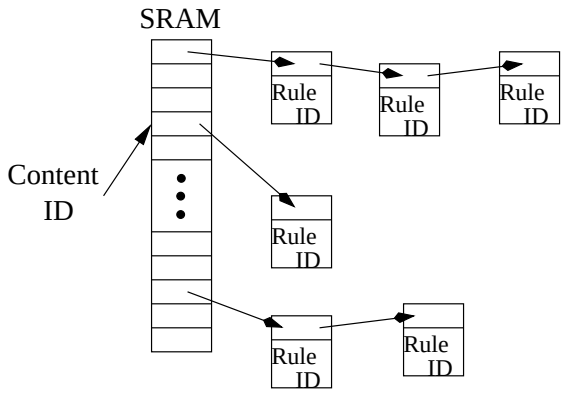

Figure 5.4: SRAM is used to maintain linked lists. Each SRAM record contains a rule ID and next pointer.

is found in four rules. Assuming an uniform distribution of signatures that match, 1.58 RIDs will be retrieved per 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, as shown in Figure 5.4. 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.

Stage 4: Header ID Mapping

Potentially matching RIDs enter stage 4, where the HID associated with the RID is determined. 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. However, the current Snort rules do not associated more than one header with a rule, so the architecture was simplified. The RID and HID are sent to stage 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. To account for header-only 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.

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 witnessed. The RID directly indexes 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. 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 seven 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.

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.

5.5.2 Efficient Context Switching

To facilitate efficient context-switching for different flows, a mechanism to save only pertinent information was 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 storage required was reduced by only storing the addresses of set bits. Unique HIDs or CIDs are stored in a buffer. When a context switch is needed, these buffers are sent to off-chip memory, and those locations are cleared in the vectors. When retrieving context information, the returned addresses are used to set the appropriate bits. To simplify the design, 512 bytes of storage are pre-allocated per flow, which can be retrieved in 32 clock cycles.

5.5.3 Buffering

Buffering was needed in this system for only two reasons. First, it is possible for a CID to be associated with more than one RID in stage 3. Hence, multiple RIDs will need to be inserted into the pipeline starting at the end of stage 3. While these are being inserted, the flow of CIDs from earlier stages is halted. Second, buffering was required to pause the pipeline when context switching was performed. During SDRAM access, new matching IDs cannot be processed.

One might assume that the system could become backlogged during the transfer of match vectors. However, consider that most matching signatures are greater than two bytes, which is all that is required to communicate that a signature matched. In the case of one byte signatures, a special byte ID is used.

The input buffer is 256 words deep, allowing for 512 different match events to be stored before back pressure to upstream modules must be asserted.

5.5.4 Control

All control of the rule processor was performed by software via the communication wrapper. There are five supported operation codes:

(1) Write SRAM bank 0 content ID to rule ID linked list mapping

(2) Write SRAM bank 1 rule ID to header ID mapping

(3) Read SRAM bank 0

(4) Read SRAM bank 1

(5) Write block RAM count value

Table 5.1: Xilinx Virtex XCV2000E FPGA resources used by the rule processor components.

Component	Function Generators	Flip Flops	Block RAMs
Stage 1	148	68	2
Stage 2	116	54	10
Stage 3	78	60	0
Stage 4	82	57	0
Stage 5	59	28	9
Stage 6	269	140	65
Stage 7	86	108	1
SDRAM Buffer 1	850	985	12
SDRAM Buffer 2	856	986	12
Communication Wrappers	1833	2050	38
Miscellaneous	687	2652	0
Design Totals	5064	7118	149

Codes (1) and (3) interact with stage 3 of the pipeline. Codes (2) and (4) interact with stage 4, and code (5) interacts with stage 6. The operation codes are described in detail in Section A.3.

5.6 Implementation Results

The rule processor has been implemented, targeted for a Virtex XCV2000E-8 FPGA, and it used 12% of the LUTs, 25% of the slices, and 93% of the block RAMs. The design synthesized at 80.6 MHz, providing a throughput of 2.5 Gbps.

FPGA features were extensively used to improve the efficiency of the rule processor. A total of 149 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 increased while the latency decreased. Due to limitations on the number of block RAMs, more rules cannot be easily supported without targeting the design to a newer FPGA with more on-chip memory.

The logic used by the system was minimal. The approximate resource requirements of each stage is shown in Table 5.1. As can be seen, stage 6 was the critical stage in terms of block RAM usage. This was due to the fact that the multiple string counts were held in block RAMs.

5.7 Discussion

5.7.1 Maintaining Throughput

The key requirement of the rule processor was to maintain system throughput over a wide distribution of packet sizes and match criteria. Clearly, if every new byte introduced by the system forces a string match, the system cannot maintain throughput since the system communicates two bytes. The number of total matches supported per packet depends on the packet size and is equal to half the number of transmitted bytes. For example, a 40 byte TCP packet can have up to 20 matching headers and strings before performance slows.

5.7.2 Effects of Context Switching

Context switching can be a bane for the throughput of the system. Consider that it takes 14 clock cycles to receive a minimum size TCP packet, while the context switch time is 32 clock cycles. Thus, a barrage of minimum length TCP packets containing matching header rules will throttle the throughput to one half the link rate.

5.7.3 Experimental Performance

The system can provide high throughput over a broad range of packet sizes. The lowest throughput of the system occurs when two minimum size TCP packets arrive belonging to different flows that contain the search term that is present in the most number of rules. For the current rule set, TCP packets with four bytes of payload containing |*00 00 00 00*| are the worst case. This results in 135 RIDs being checked. The average-case and worst-case throughput of the rule processor for this case is plotted versus the number of payload bytes in Figure 5.5. Note that as packet size increases, the throughput of the system increases to the maximum value.

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

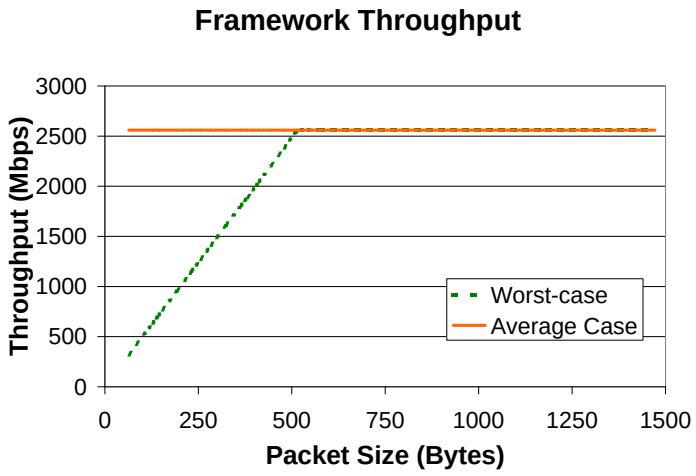

Figure 5.5: Average-case and worst-case throughput of the rule processor versus the input packet size.

5.7.4 Memory Bandwidth Requirements

Access to SRAM has been deeply pipelined, allowing a memory look-up from each bank on every cycle. Thus, the SRAM banks can provide a maximum bandwidth of 5.76 Gbps.

It is more difficult to efficiently utilize SDRAM due to the unknown nature of traffic patterns. Not every packet that arrives in the system may force a SDRAM event. It is most undesirable to stall the rule processing pipeline. In order to prevent stalling, SDRAM fetches context information the instant information from a different flow arrives, allowing SDRAM latency to be masked with the processing of the previous packet.

A worst-case traffic pattern requires that every packet retrieve context for the given flow and store the updated context back upon the end of the packet. This requires the transfer of 1024 bytes of information per packet. Assuming that all packets are minimum sized TCP packets, nearly eight million packets per second can be received. Such a traffic pattern requires 65 Gbps memory bandwidth, more than six times the available memory bandwidth of the FPX platform.

Memory bandwidth to handle the context storage is a bottleneck. The solution is to change how context is stored for each flow. To simplify the buffering and context switch logic, 512 bytes of memory were pre-allocated for each flow, allowing storage of 224 matching headers or signatures and 24 potentially matching rules at one instance, as shown in the SDRAM record of Figure 5.6.

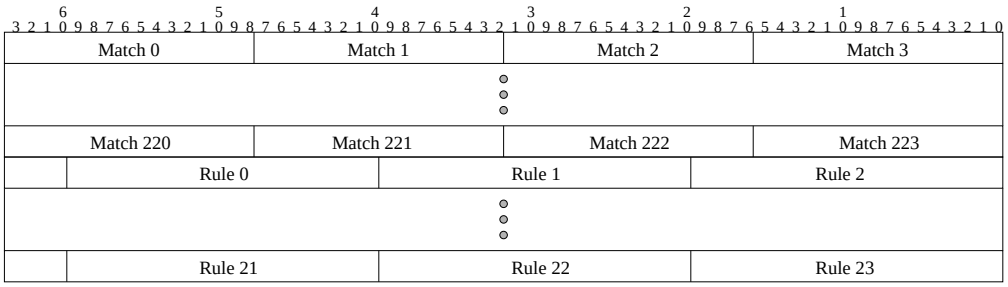

Figure 5.6: 512 bytes of storage were pre-allocated per flow, which allows for storing up to 224 headers or signatures and 24 potentially matching rules.

However, this may waste space in some cases. Improvements could be made to the memory controller to only send the data that needs to be stored. The unknown nature of how many header and signature matches occur per flow made it difficult to allocate buffer space.

5.8 Observations

5.8.1 Memory

To perform all rule processing in a hardware device, the memory subsystem requires a large bandwidth. The RPF performed burst operations to achieve high throughput. However, even this implementation could not handle the absolute worst-case traffic patterns. Assuming a burst of minimum length TCP packets, the most amount of data that can be held for context is 82 bytes in the FPX platform.

In order to accommodate the worst-case traffic, the system would benefit from a memory device with faster throughput. This system uses 100 MHz SDRAM, but there are memory elements available now that have higher clock frequencies as well as transfer data on both the rising and falling edge of the system clock (double data rate SDRAM).

Chapter 6

Results

This chapter presents the results of the three architectures. The test environment is briefly described, and the chapter is concluded by analyzing the results and performing comparisons. The comparisons are not only between architectures but also examine components implemented by other researchers.

6.1 Testing Environment

All experiments were performed in the Applied Research Laboratory (ARL) [5] at Washington University.

6.1.1 FPX

All architectures were tailored for use with the Field programmable Port eXtender (FPX) platform [67, 69] of Figure 6.1. The FPX consists of two FPGAs. The first is the Network Interface Device (NID), and the second is the Reprogrammable Application Device (RAD). The NID is a Xilinx Virtex XCV600E FPGA, and the RAD is a Xilinx Virtex XCV2000E FPGA. The architectures described in this thesis were synthesized into the RAD.

The NID routes information between the RAD and the downward and upward interfaces of the FPX over an Utopia bus. The FPX can interface with additional FPX cards, with a switch, or with various line cards. Control of the NID is performed by a software tool called NCHARGE [112] that programs VPI/VCI routes. Additionally, programming of the RAD FPGA is performed via the NID, which receives the FPGA bitfile from NCHARGE.

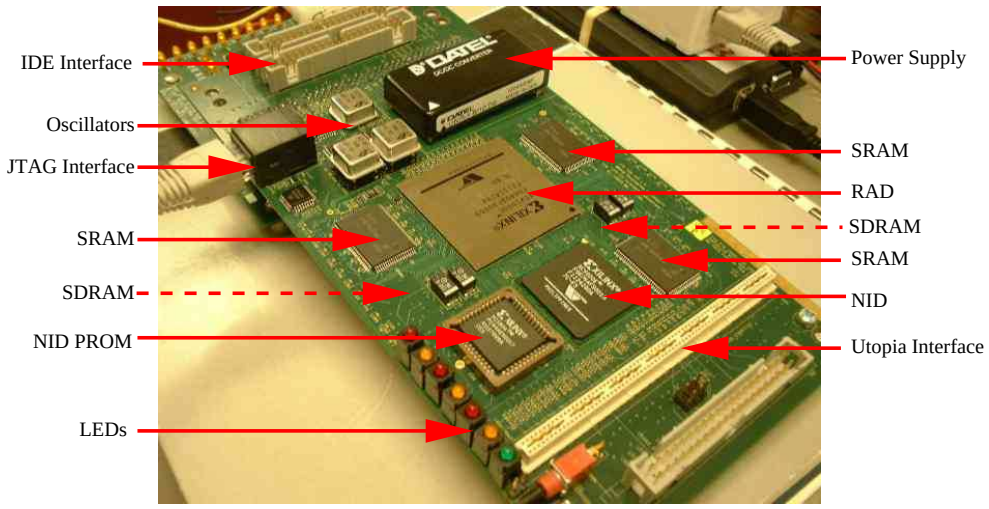

Figure 6.1: The FPX has two FPGAs, three banks of SRAM, and two banks of SDRAM (on the reverse side of the card).

The RAD is the main data processing FPGA, and it has access to two 2 MB ZBT SRAMs and two 512 MB SDRAMs. The RAD also has two Utopia interfaces: the switch interface and the line card interface. The NID forwards ATM cells, to one or both interfaces of the RAD. Once processed, the RAD forwards the cells back to the NID to be re-injected into the network.

The main communication unit of the FPX is a 53-byte ATM cell, padded to be transmitted in 14 32-bit words (56 bytes). IP packets can be transferred over ATM by using the ATM Adaptation Layer 5 (AAL5) protocol [48]. The current generation FPX can operate at 100 MHz, providing a maximum data throughput of 3.2 Gbps.

6.1.2 GVS-1500

The GVS-1500 [6], as shown in Figure 6.2, is an environment that leverages the FPX platform. There are two stacks of networking devices in this configuration. One stack consists of a line card and one or more FPX cards. The other stack consists of a line card and a NID pass-through card (NID-PT). The NID-PT de-encapsulates control operations from a higher level transfer protocol to configure the FPX cards.

This system can be placed in a wiring closet to act as a gateway between the internal network and the external network. This system is well suited for an intrusion sensor operating in active mode because traffic can flow in through one stack and out the other.

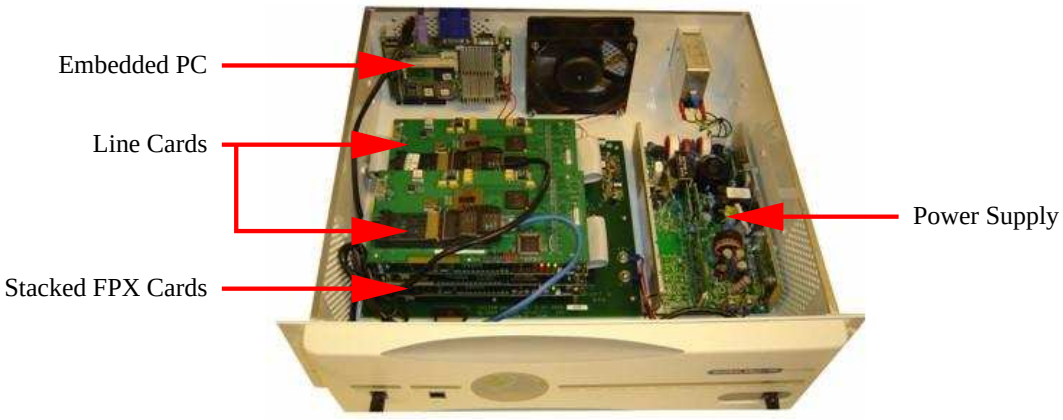

Figure 6.2: The GVS-1500 uses FPX cards in a stacked configuration with Gigabit Ethernet line cards at the top of the stack. An embedded PC can be used for control.

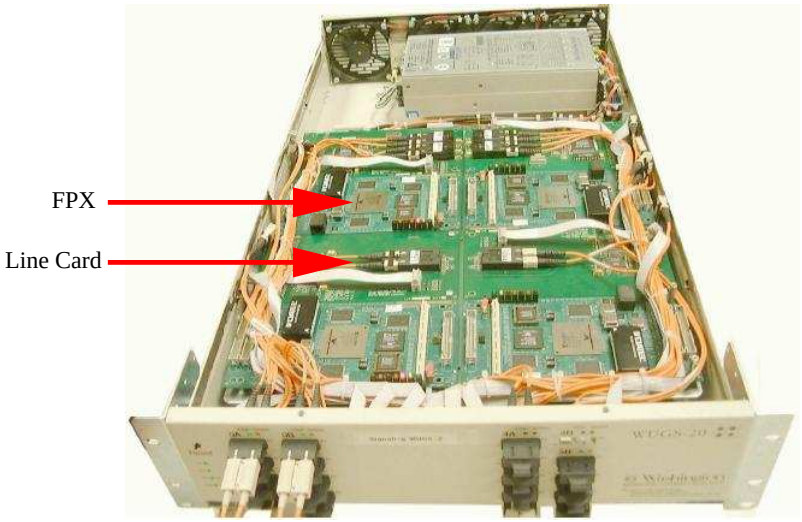

Figure 6.3: The WUGS-20 provides eight processing ports.

6.1.3 WUGS-20

The Washington University Gigabit Switch (WUGS) [122] is the second environment that leverages FPX technology. The WUGS-20 is an eight-port switch that allows the FPX to be plugged directly into one of the switch ports. Routes in the WUGS-20 are fully programmable, allowing data to be processed by each port element if desired. In addition, the WUGS-20 supports multicast. Figure 6.3 shows the WUGS-20 configured with four FPX cards and four line cards.

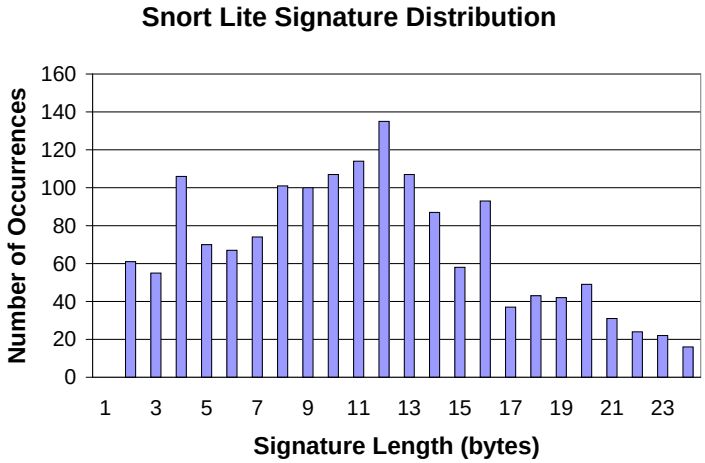

Figure 6.4: The programmable signature distribution for Snort Lite.

6.2 Snort Lite Results

6.2.1 Compatible Snort Rules

Of the 2,464 Snort rules in the database, Snort Lite can be configured to support 1,580 (65%) of the rules by programming 1,762 separate rules (one for each signature) and relying on software to determine that an actual Snort rule matched when multiple signatures were present. This percentage was determined by counting rules that contain the first occurrence of signatures in the range 1 to 32 bytes. However, since Snort Lite searched for signatures that were between 2 and 24 bytes, only 1,451 of the 1,580 rules and 1,599 of 1,762 signatures were supported at one time. This range covers the largest group of signatures. Snort Lite could not support header-only rules.

The distribution of signatures is shown in Figure 6.4. The engine with the most number of signatures was the engine that scanned for 12 byte signatures. While there were more unique 4-byte signatures, these 4-byte signatures were also used in rules multiple times, so the rules with additional instances were discarded. The maximum false positive probability was 2.74E-10, or one in four billion.

6.2.2 Throughput

The maximum attainable throughput of the system is calculated by multiplying the number of bits that can be processed per clock cycle and the frequency of operation, which results in 502 Mbps (8 bits * 62.8 MHz).

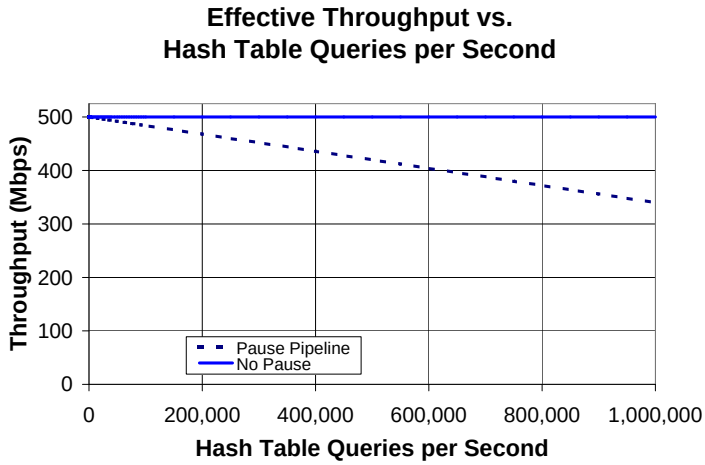

Figure 6.5: The throughput of the system degraded as the number of hash table queries per second increased. This was due to the fact that the byte pipeline paused until the match resolved. The system remained at 80% capacity when 600,000 attack packets were processed.

The expected throughput of the system, however, depends on the traffic pattern and characteristics. Because data arrives four bytes per clock cycle, a sudden long burst of back-to-back traffic can cause the system to assert back pressure.

Additionally, the matching rules found in packets forces pipeline stalls that degrade throughput. The choice to pause the pipeline to allow memory transactions to complete adversely affects the throughput of the system, as shown in Figure 6.5.

This NIDS has sufficient performance to be robust against denial of service attacks. On a fully utilized 500 Mbps link, up to 1.5 million packets can arrive per second. Even if 600,000 of these packets contain search strings, the throughput of the system is only reduced to 80% capacity.

6.2.3 System Tests

Testing was performed by placing the system between a bank of computers and an external network, as shown in Figure B.1. The device was tested in active mode with data known to cause rule matches. Rules were loaded into the device using a web-based graphical user interface (GUI) while traffic was sent through the system. With one test, a web page that contained 100 occurrences of the search string *George* was fetched, causing the hardware matching circuit to trigger 100 times and to generate an alert packets. Alerts arrived at the software controller and were displayed with a

Table 6.1: A summary of the resources available on various Xilinx FPGAs.

Device	Slices Available	Block RAM (Kbits)	Distributed Memory (Kbits)
XCV2000E	38,400	640	600
XC2V6000	33,792	2,592	1,056
XC4VFX100	49,152	4,320	768

Java-based graphing application, as shown in Figure A.17. By staging a set of data online, each signature length and header field type was tested.

In order to push a heavy traffic load through the system, *wget* was used to recursively fetch web pages from `http://www.cnn.com` and `http://www.whitehouse.gov`. Rules were programmed that contained the search criteria *George*, *Politics*, and other news of the day. Multiple simultaneous connections were created in this way, and multiple rules were seen to match. Bulk file transfers were performed in the background to expose the system to traffic rates near 500 Mbps.

6.2.4 Next Generation FPGAs

The circuits designed would benefit by implementation in newer FPGA technology such as the Virtex II and Virtex 4. These devices have more resources than the Xilinx Virtex XCV2000E. The resources in these FPGAs are compared in Table 6.1.

In Figure 6.6 the relative improvements available by targeting the design to newer FPGAs are shown. With an increased amount of block RAM, more Bloom engines are supported, allowing for the number of rules supported to increase. The frequency improvement is approximately 1.5x, but the throughput is considerably better in newer FPGAs. By duplicating the match logic, more bytes per clock cycle can be processed. The Virtex II can instantiate two parallel copies of 55 Bloom engines (one for each signature length being scanned for), while the Virtex 4 can instantiate four parallel copies since it has more block RAMs.

6.3 SIFT Results

6.3.1 Compatible Snort Rules

Of the 2,464 Snort rules in the database, the SIFT architecture supported 2,311 of the rules by programming 1,995 signatures into the system. SIFT also supported the

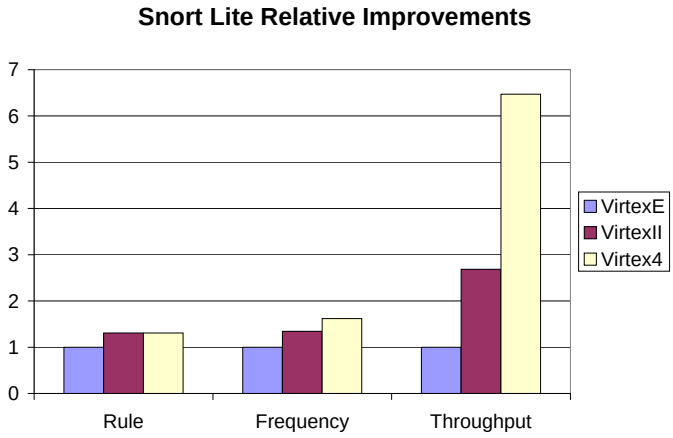

Figure 6.6: The projected relative improvements of Snort Lite using newer FPGAs.

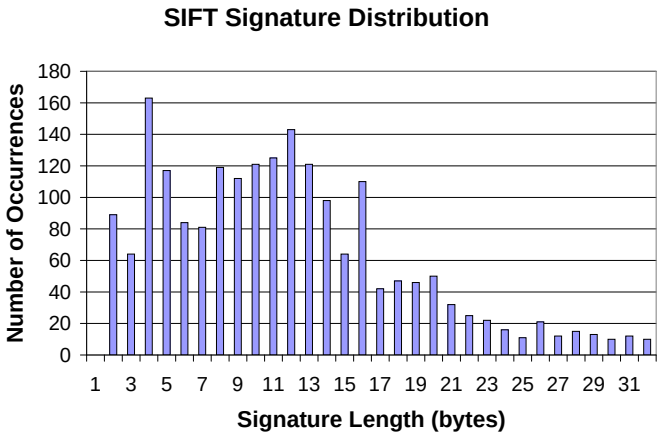

Figure 6.7: The number of signatures associated with each length for SIFT.

168 header-only rules. The distribution of signature lengths supported are shown in Figure 6.7.

6.3.2 Throughput

The maximum throughput of the system was 2.5 Gbps. High throughput can be maintained even with a large frequency of matches. Even though SRAM look-ups can be obtained in back-to-back cycles, throughput degradation can occur when multiple Bloom engines match over an extended period of time. Figure 6.8 shows the allowable number of matches for various packet lengths before performance degradation occurs. Figure 6.9 shows how the throughput of the system decreases if too many matches occur per second. Just under 100 million matches have to occur per second for the throughput to degrade at all.

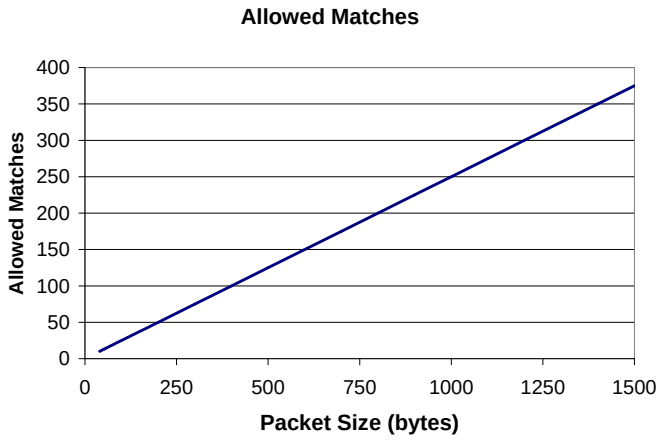

Figure 6.8: The allowable matches that can occur within a packet of a given length without causing throughput degradation.

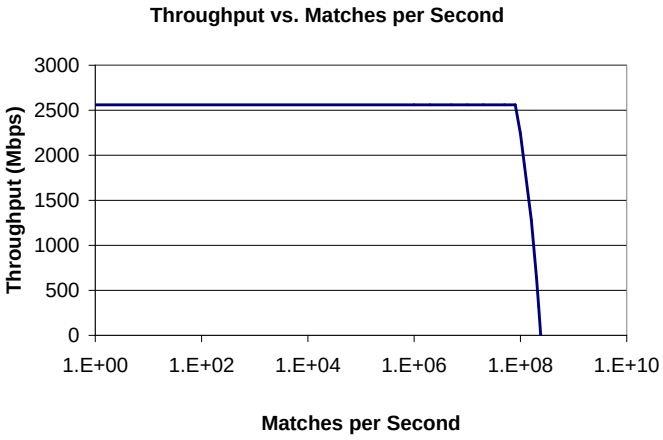

Figure 6.9: The throughput of the system versus the number of matches per second. The system could sustain 100 million matches per second at maximum throughput.

6.3.3 SIFT Test Configuration

SIFT was configured to use Bloom engines to scan for 4, 6, 8, 10, and 12 byte signatures. All 4 and 5 byte signatures were loaded in the 4 byte engine, 6 and 7 byte signatures in the 6 byte engine, and so on. All signatures greater than or equal to 12 byes were placed in the 12 byte engine. Table 6.2 summarizes how signatures found in Snort version 2.2 were loaded into the hardware. Non-unique signatures refers to the signatures that were truncated to the appropriate length, but the truncated version had already been programmed into the engine.

The Bloom engine that scanned for 12 byte signatures had the highest false positive probability, which was 2.1E-4 or 1 in 5000. The next highest false positive

Table 6.2: A summary of how the Bloom engines were loaded.

Scan Range	Signatures Loaded	Non-unique Signatures	False Postive Probability
4-5	262	18	4.3E-8
6-7	163	2	1.2E-9
8-9	227	4	1.5E-8
10-11	244	2	2.5E-8
12+	872	137	2.1E-4

probability occurred in the engine that scanned for 4 byte signatures. The false positive probability was 7.1E-8, or 1 in 150 million.

The system was configured, as shown in Figure B.3, to monitor to monitor all traffic entering or leaving Washington University's 19,000-node campus network and process the current Snort rule database. The throughput witnessed varied with time, but it peaked at 350 Mbps.

6.3.4 Testing Results

System events were monitored, and the results were displayed using MRTG. The graphs plot the five minute average of events per second. Figure 6.10 shows the total incoming TCP packets, and Figure 6.11 shows the amount of TCP packets that were sent to the intrusion PC during the week of February 13-20, 2005. Figure 6.12 shows the reduction in data sent to the host PC for TCP traffic. Similar figures for UDP, ICMP, and other IP packets are shown in Figure D.1, Figure D.2, and Figure D.3, respectively. As can be seen, SIFT reduced the amount of TCP traffic that needed to be examined by the Snort PC by 85% on average. The traffic reduction achieved based on protocol is summarized in Table 6.3.

Table 6.3: A summary of the reduction in packets transmitted to the Snort PC for the week of February 13-20, 2005.

Protocol	Incoming Packet Average	Outgoing Packet Average	Reduction
TCP	28.2 K	2.71 K	90%
UDP	0.50 K	0.07 K	86%
ICMP	1.03 K	0.87 K	15%
IP	0.34 K	0.01 K	96%

The number of matching headers over the given time period is shown in Figure 6.13. Matching headers significantly dropped off late in the morning, even though the number of packets traversing through the system increased at this time. The number of matching headers ramped back up early in the afternoon, only to decline again until evening. This was a periodic event that occurred on multiple days of observation.

The average number of matching 4-byte signatures per second is plotted in Figure D.13. Although it was expected that the number of matching signatures would scale with the traffic load, the time of day had nearly no impact on the number of matching signatures. This was true for all Bloom engines, as shown in Figures D.14-D.17 in Appendix D. Also, the number of matches for each QBF engine was approximately the same, even the 12-byte QBF. This indicates that longer signatures do not occur frequently.

The Snort PC was a 2.13 GHz AMD Athlon MP 2600+ with 3 GB or RAM running Red Hat Linux. Snort had no trouble keeping up with the average 40 Mbps of traffic that was passed to it for processing. When the same PC was booted into Windows XP, only 50% of the traffic was inspected. The implementation of Snort for the different operating systems apparently has a profound impact on performance.

Using the Linux version of Snort to process the traffic exiting SIFT, an average of 5% of the traffic resulted in matching rules, indicating that it would be possible to further improve the hardware to reduce the forwarded traffic. Of the traffic that was forwarded, 78%, 16%, and 6% of the resulting matching Snort rules were for ICMP, TCP, and UDP packets, respectively, as shown in Figure 6.14. It appears worthwhile to add a better hardware filter for ICMP traffic.

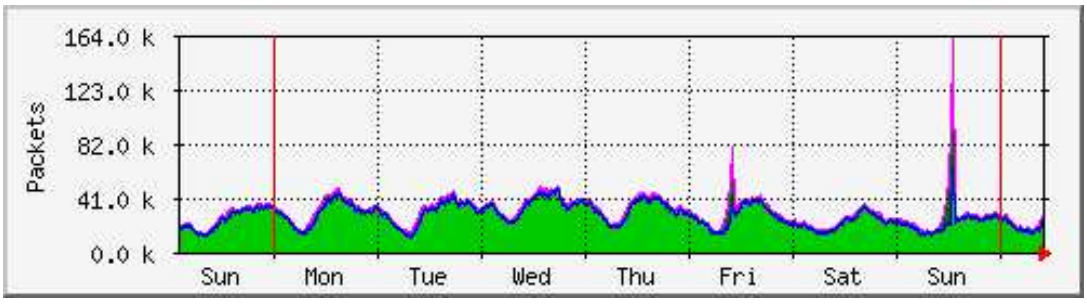

Figure 6.10: The number of incoming TCP packets versus time on Feb 13-20, 2005.

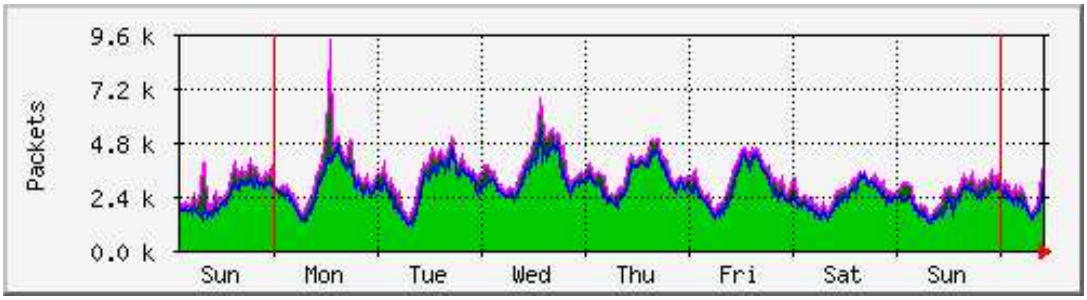

Figure 6.11: The number of outgoing TCP packets versus time on Feb 13-20, 2005.

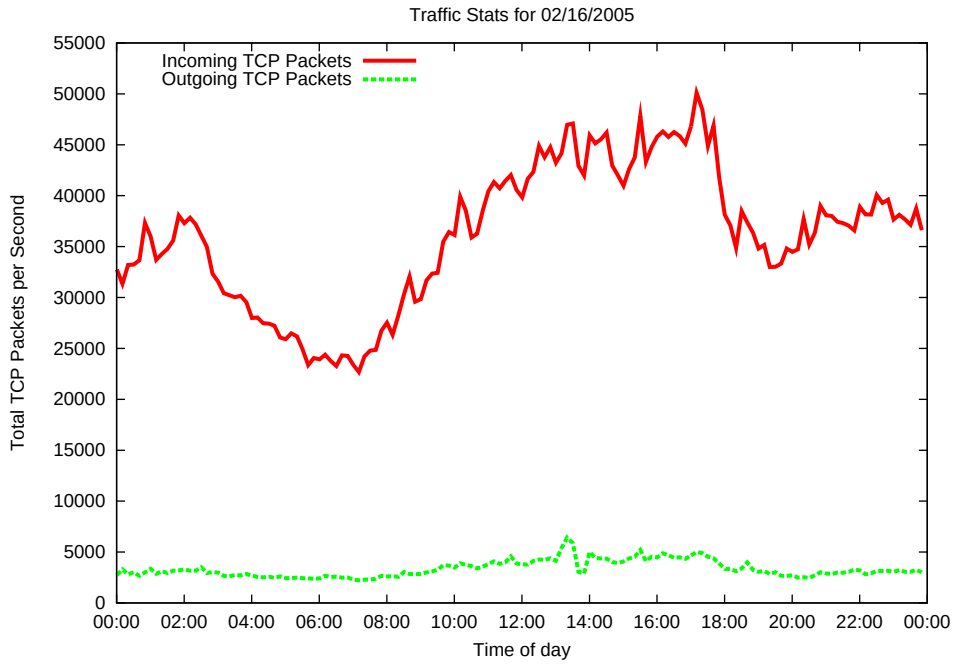

Figure 6.12: The number of incoming and outgoing TCP packets versus time on Feb 16, 2005.

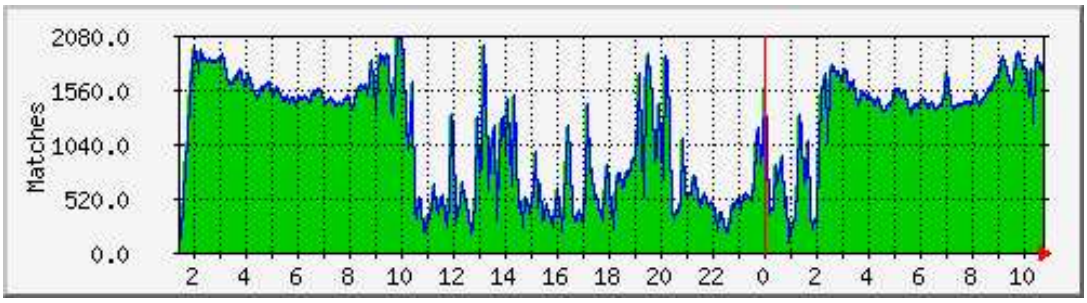

Figure 6.13: The number of matching header-only rules versus time on Feb 16-17, 2005.

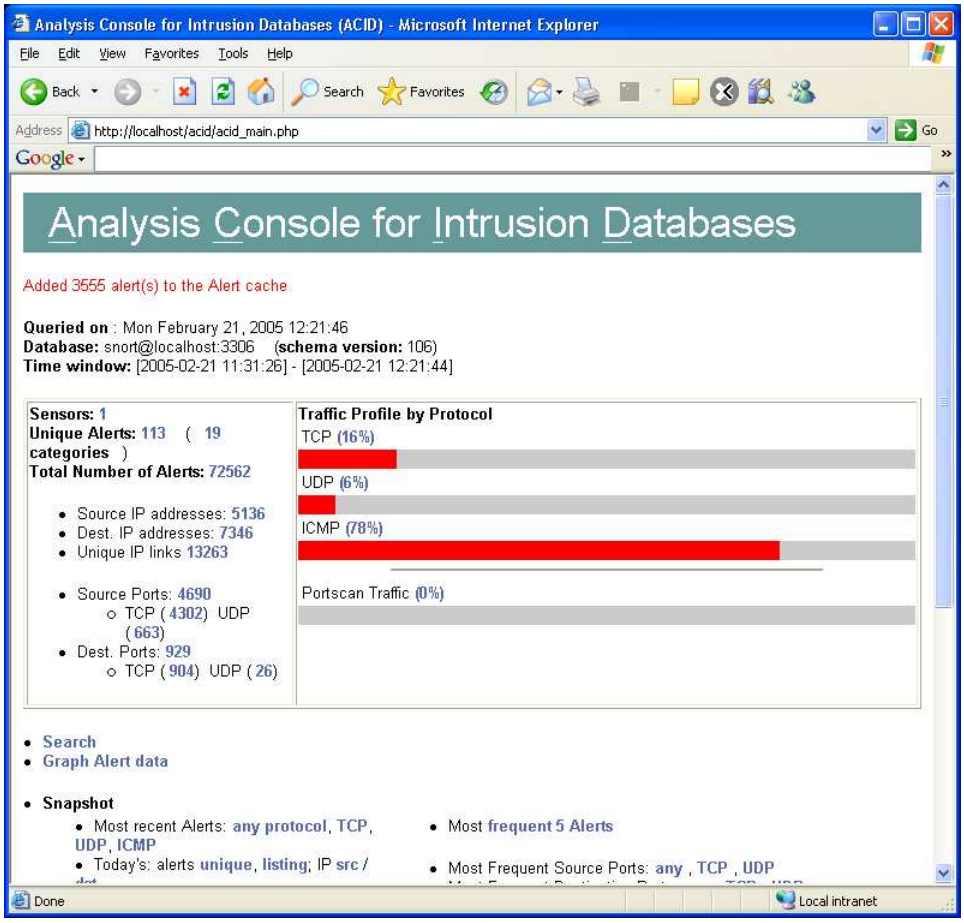

Figure 6.14: Using an analysis console, the breakdown of alerts by protocol can be known.

6.4 Rule Processor Results

6.4.1 Compatible Rules

The rule processor can be used to process a wide range of network intrusions. Signatures in the payload can be fixed strings or regular expressions. Additionally, the header type can be a complicated expression involving multiple fields in the packet's header. The rule processor can support all potential Snort rules by representing these elements as 16-bit integers.

6.4.2 Throughput

As in all the architectures, the number of matches that can occur per packet is limited to a certain value before the system experiences throughput degradation. This is a

function of the packet size, and it is shown in Figure 6.15. For full-length Ethernet frames, the sum of header matches and content matches can be 750. Past this point, more data is communicated than arrived.

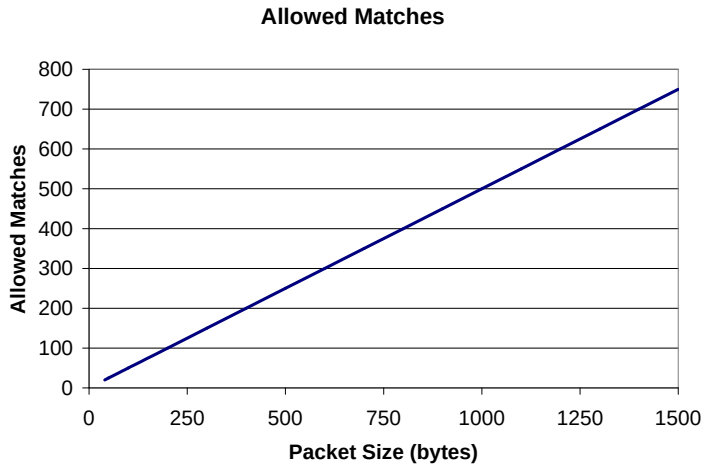

Figure 6.15: The allowable matches versus packet length before throughput degradation.

The rule processor could impose additional throughput degradation for two reasons. Context switching was an expensive operation, requiring the retrieval of 512 bytes from SDRAM as well as flushing the processing pipeline. The number of context switches that can be performed before performance is hindered is shown in Figure 6.16.

The second reason was due to rule look-ups being inserted. The pipelined nature of the rule processor allowed potential rules to be processed back-to-back with no pauses. A large number of rule look-ups can be performed before the system must stall the flow of packets through the system, as shown in Figure 6.17. Performance degradation begins when the number of lookups required exceeds 80 million per second. This number assumes no context switching is performed.

Latency

Latency was not a critical issue for this system due to the feed forward nature of the processing flow. One module processed a packet before that packet was passed to the next module for additional processing. The rule processor was the last to receive the packet, and by this time all relevant matching information was communicated. The rule processor could then make a decision on what action to take on the packet.

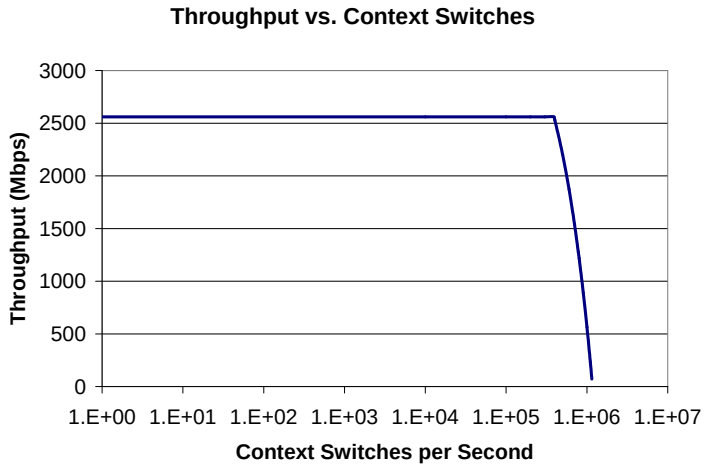

Figure 6.16: The throughput of the system versus the number of context switches performed per second.

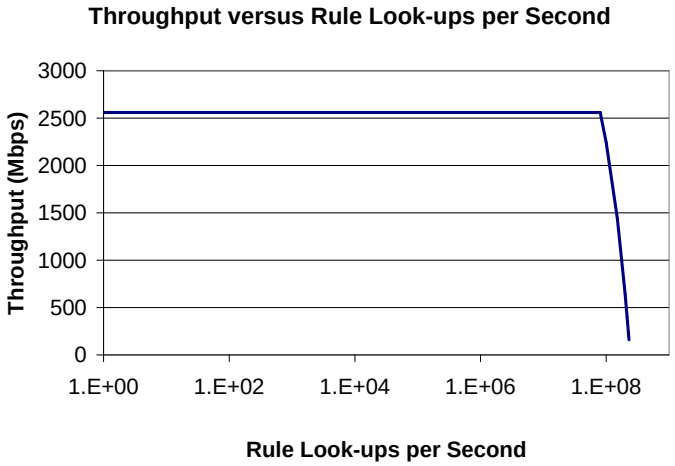

Figure 6.17: The throughput of the system versus the number of matches per second.

6.4.3 Next Generation FPGA Projections

Using newer technology, more rules can be stored and higher throughput can be achieved, as shown in Figure 6.18. With access to more block RAM, the number of rules supported can be increased by a factor of five while also increasing the throughput. The throughput is increased by duplicating the processing pipeline, allowing more match IDs to be processed per clock cycle. With a Virtex 4, the rule processor could operate at 15.9 Gbps.

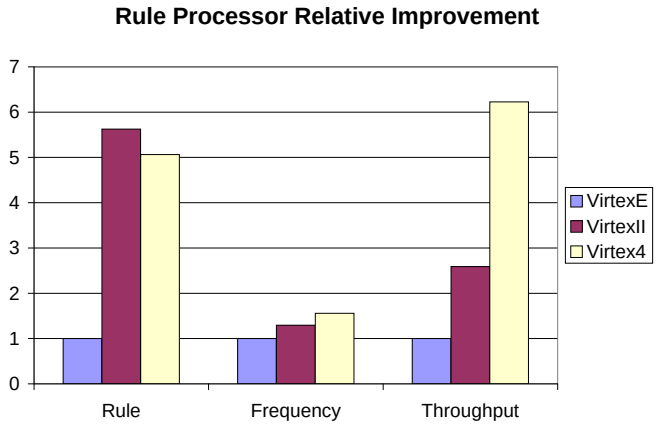

Figure 6.18: The projected relative improvements of the rule processor design using newer FPGAs.

6.5 Comparisons

6.5.1 Hardware vs. Software

The bandwidth reduction provided by SIFT begs the question whether a full-featured implementation of Snort in hardware is necessary. SIFT reduced the traffic that needed to be examined by Snort to less than 15% the total traffic. Snort running on a PC was capable of processing all this traffic.

However, as link speeds increase, the reduction, while still substantial, surpasses the capabilities of the general purpose computer. Current processors have a Gigabit NIC attached. Any traffic load above this is not supported.

6.5.2 Effects of Context Switching

The effect of flow-based context swapping was detrimental to throughput. Under worst-case traffic patterns (back-to-back interleaved 40-byte packets), the FPX platform had enough external memory bandwidth to store 82 bytes of state information. In the RPF, this means storage for up to 20 different matching criteria. Whether this is enough for TCP flows is yet to be determined. The number depends largely on the rule set being processed and the characteristics of the traffic.

Table 6.4: 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.

Group and Component	Device	Logic Cells	Throughput
GaTech Stream Assembler[81]	Virtex 1000	876 (10%)	3.2 Gbps
NU Flow Monitor [82]	VirtexII-8000	-	48.3[1] Gbps
WashU TCP Processor [96]	Virtex4 FX140	22,100 (35%)	10.3 Gbps
WashU BV-TCAM [108]	Virtex4 FX100	4,200 (10%)	10 Gbps
Crete Pre-decoded CAMs[111]	VirtexII-6000	64,268 (95%)	9.7 Gbps
GaTech Decoder Trees [28]	VirtexII-8000	54,890 (81%)	7 Gbps
Tokyo Trie-based Hash [115]	VirtexII-6000	2,365 (7%)	10 Gbps
UCLA Packet Filters [24]	Spartan 3 2000	15,202 (37%)	3.2 Gbps
USC Partitioning [19]	VirtexII Pro 100	15,010 (15%)	4.5 Gbps
WashU Bloom Filters	Virtex4 FX100	35,850 (85%)	20.4 Gbps
WashU Rule Processor	Virtex4 FX100	40,200 (95%)	15.9 Gbps

6.5.3 Comparison with Related Work

Components of network flow processing circuits can be implemented efficiently in reconfigurable hardware. Table 6.4 shows projected resource requirements for components synthesized 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.

[1]This assumes 40 Byte packets.

Chapter 7

Conclusion

Network intrusion detection and prevention systems have become an essential part of the Internet infrastructure. Attacks from worms, viruses, and other malware pose very serious risks to networked systems. An IDPS has two primary purposes. First, it must detect all potentially damaging events that can occur in a system. Second, it must prevent the harmful activity from being performed by blocking the flow of harmful traffic into the system.

Snort, the most popular intrusion detection system, allows an administrator to write rules that describe what constitutes an intrusion. These rules can be customized for specific networks with particular characteristics.

A rule processing system should be adaptive, allowing new rules to be added quickly and efficiently. The system must operate in real-time, allowing actions to be taken to stop attacks and prevent damage from occurring. Finally, a rule processor must provide useful information to system administrators that can be used to combat security breaches.

7.1 Summary of Approaches

In this thesis, three architectures to perform rule processing in reconfigurable hardware were presented. Such hardware-accelerated systems are becoming necessary as link speeds increase to gigabits per second and beyond.

The first architecture, Snort Lite, used a single FPGA to perform the major components of rule processing, header processing and string matching. Correlation between criteria was performed by retrieving header rules associated with signatures when signatures matched.

The second architecture, SIFT, filtered harmless network traffic destined for a general purpose processor running intrusion detection software on a PC. Over 85% of traffic did not contain criteria that would cause a rule match. Additionally, the full-feature set of Snort could remain on the processor, where it is easier to implement.

The third architecture developed was the rule processing framework, which allowed the addition of modules to perform the components of rule processing. The correlation of search criteria in the rule processor was described. Processing blocks, including a header rule processor, a payload scanner for regular expressions, and a payload scanner for static signatures, were developed to communicate information about what criteria matched in the packet or flow.

The results of the three architectures are summarized in Table 7.1. Snort Lite supported the least Snort rules and had the lowest throughput, while the rule processor supported all Snort rules and achieved the highest throughput on the FPX.

Table 7.1: A summary of the results for the three architectures using the FPX platform. The target FPGA device was the Xilinx Virtex XCV2000E.

Architecture	Compatible Snort Rules	Throughput (Gbps)	LUT Usage	Slice Usage	Block RAM Usage
Snort Lite	65%	0.52	55%	77%	99%
SIFT	93%	2.56	58%	84%	96%
Rule Processor	100%	2.56	12%	25%	93%

7.2 Contributions

In this thesis, architectures were developed to perform rule processing in hardware. After characterizing the current Snort rule database, it was found that the bulk of Snort rules contain a single header rule and a single signature. The architectures were optimized to perform well in this case. However, since not all of the rules were of this form, additional circuits were developed to support the more general processing of more complex rules.

The following elements were created during the course of this thesis work.

First, Bloom filters have been successfully mapped for use in a FPGA. Previous work in [37] showed how to map Bloom filters into a FPGA. This information was used to create a Bloom filter component that is reliable, robust, and adaptable.

Second, a robust communication component called the communication wrapper was created. The wrapper provided an intuitive and simplified interface for communicating between hardware and software.

Third, other reusable infrastructure was developed. Hash tables, context engines, and alert generators were developed to store/retrieve data and communicate system information to software. Details of these components are discussed in Appendix C.

Fourth, the first high-scale content scanners that operated on TCP flows was developed. Previous work by [96] had performed string matching on TCP flows for just four signatures.

Fifth, a scalable, high-throughput framework to provide rule correlation in hardware was developed. The rule processing framework took flow and match information from processing modules and determined matching rules.

Finally, traffic analysis on a 19,000 node network was performed. The bandwidth of this large network did not overload the system. As the SIFT architecture showed, over 85% of the traffic being monitored was benign, containing no matching search criteria from version 2.2 of the Snort database. This suggests that general purpose processors with traffic off-loaders implemented in hardware could be used today to significantly improve the performance of network monitoring systems.

7.2.1 Conclusions Drawn

Much was learned through development of the network processing circuits. Snort Lite was capable of supporting many rules and was relatively compact. However, it was also too slow for modern networks and lacked support for 35% of current Snort rules. SIFT improved on Snort Lite by parallelizing the string-matching computation, allowing multiple bytes to be processed per clock cycle. Additionally, SIFT performed matching for large numbers of signatures within TCP flows. However, SIFT required custom logic for header-only rules and supported a limited range of signature lengths. The rule processing framework solved these problems, providing the flexibility to support all rule types and added multiple features. It allowed component designers to focus on improving techniques to perform header processing and payload scanning without needing to correlate the two.

Much of the difficulty in performing rule processing in hardware was not the constituent component processing, but rather in efficiently identifying that a rule

matched. For example, it was harder to report which signature matched than to just report that a matching signature was found. The architectures developed for this work returned an ID value associated with the matching criteria.

One of the most surprising results was the ease of implementation of header-only processing. A brute-force compare on the appropriate fields in packet headers was not as resource-excessive or computation-heavy as originally assumed. The method scales to support header processing for the set of Snort rules using only a minimal amount of FPGA resources. While counter-intuitive, the results indicated that CAMs may not be needed for systems that use FPGA logic to perform Snort rule processing.

7.3 Future Work

All of the architectures presented were well suited for use with anomaly detection and automatic rule generation systems. Systems, like [71], can be used in tandem with these architectures to detect malicious traffic, create a rule, and load the rule into the system.

Porting these architectures to newer generation FPGAs to take advantage of additional resources and benefit from faster clock frequencies is a logical next step for expanding these designs. Systems with such architectures can significantly outperform existing commercially available systems.

Finally, alternate techniques to correlate matching headers and signatures could be explored. Scalable techniques are desired for future rule processing systems to handle thousands of header, payload, and cross-flow rule permutations.

Appendix A

Control Software

All of the architectures do not come pre-programmed with any rules (except for the SIFT architecture which has logic for 168 header-only rules). As a result, a software controller must program the circuits with rules to process. The software controllers for the three architectures are similar, but all have unique functionality.

A.1 Snort Lite

The software developed for Snort Lite consisted of three main files:

- `GenHashValues.java`
- `RuleSetter.java`
- `Sender.java`

GenHashValues creates the coefficients for the hash functions used by the Bloom filter engines (refer to Section 3.4.2). To run the program, the following command is executed:

```
$ java GenHashValues <# hash functions> <max hash value>
<# hash bits> <max string length> <output filename>
```

The number of hash functions parameter specifies how many different sets of coefficients are to be created. The maximum hash value is the depth of the vector to be indexed. The number of hash bits specifies how many bits the address will be. Note that the current implementation requires that this be a multiple of four. The maximum string length specifies the maximum number of bytes supported. Finally,

the output filename is the name of the VHDL file to create. All fields are required. For Snort Lite, the following command was run:

```
$ java GenHashValues 8 4096 12 32 hash_values.vhd
```

The *RuleSetter* file converts Snort rules to UDP control packets. To run the program, use the following command:

```
$ java RuleSetter <Coefficients File> <Rule File> <Output File>
```

This program reads a coefficients file (*.vhd*) and a rules file (*.rules*) to create a *.pay* file that is read to send the configuration information to the hardware. Rules in the *.rules* file are exactly as they appear in the Snort database. For example, the following rule would result in the payload file shown below.

```
alert tcp $EXTERNAL_NET any → $HOME_NET 79 (msg:"FINGER cmd_-
rootsh backdoor attempt"; flow:to_server,established; content:
"cmd_rootsh"; classtype:attempted-admin; reference:nessus,10070;
reference:cve,CAN-1999-0660; reference:url,www.sans.org/y2k/-
TFN_toolkit.htm; reference:url,www.sans.org/y2k/fingerd.htm;
sid:320; rev:6;)
```

```
AnalyzerPayload
70090000
6f747368
645f726f
0000636d
01400000
HeaderPayload
76000000
01400610
c0a80000
FFFF0000
c0a8c802
FFFFFF00
0000FFFF
004f004f
```

```
BloomPayload
74070000
28001f59
28000db5
28020c61
28020e47
2804095d
28041af1
280608e9
2806054d
```

Each rule generates sections for each of three operation codes. The analyzer payload programs the string into the hash table analyzer. The header payload adds the header rule to the header processing block, associating it with the signature ID. The Bloom payload block consists of eight words, one for each of the bits to set in the bit vector for the Bloom engine that scans for ten byte signatures.

The *Sender* program communicates with the hardware. All operation codes can be specified as options. To use this program, the following command is executed:

```
$ java Sender [-h IP Address] [-p Control Port] [-f Payload
Filename] [-t Test Outgoing] [-c Statistic Number] [-d Header
Entry] [-a Alert Destination] [-r Reset System] [-m Test Message]
```

The -h options specifies where to send the control packets. The -p option specifies the control port that the hardware is listening on. The -h and -p options are required for all cases. The remaining options specify particular operation codes.

The -f options loads rules specified in *.pay* files, into the hardware. The -t checks if the hardware is operational. The -c option requests that a system event counter be queried. The -d options specifies reading a header rule from SRAM. A header rule is specified via a signature ID. The -a option changes the destination IP address and port of the alert messages sent from the system. This option has two parameters. The first is the IP address to use and the second is the port. The -r option clears all rules from the system. This is available so that a hard reboot of the hardware is not required. The -m option allows for a simple test packet to be sent through the hardware to test that a new rule has been adopted. An UDP packet will be created with the string that follows -m.

The packet formats used by the various control messages are shown in Figures A.1 through A.5. Figure A.1 shows the format for adding or deleting signatures from the analyzer. Figure A.2 shows the packet format used to set/reset bits in a Bloom filter. Figure A.3 shows the packet format for adding a header entry to SRAM. Figure A.4 shows the format used to change the alert message destination IP address and port. Finally, Figure A.5 shows the format used to remove a header entry, read a header entry, or read a statistic from the hardware. Opcode x78 deletes a header rule, opcode x82 reads a header rule, and opcode x84 reads a system counter.

As an example of how the system is loaded once booted, the following list of commands are executed:

```
$ java Sender -h 192.168.50.2 -p 48879 -a 192.168.50.15 3000
$ java Sender -h 192.168.50.2 -p 48879 -f test.pay
$ java Sender -h 192.168.50.2 -p 48879 -c 00
```

Alert messages that return from the hardware can have five types. All communication from the hardware uses UDP packets. Type 1 is used to confirm that the opcode sent was successfully adopted by the hardware and has the format shown in Figure A.6. Type 2 is used to return a header entry, and it is shown in Figure A.7. Type 3 is used to report a statistic and follows the format of Figure A.8. Type 4 is used in the event of a rule match and has the format of Figure A.9. For this type of alert, only the matching rule ID number and the header of the packet are returned. Finally, type 8, as shown in Figure A.10, is used to return the entire packet to software for further inspection.

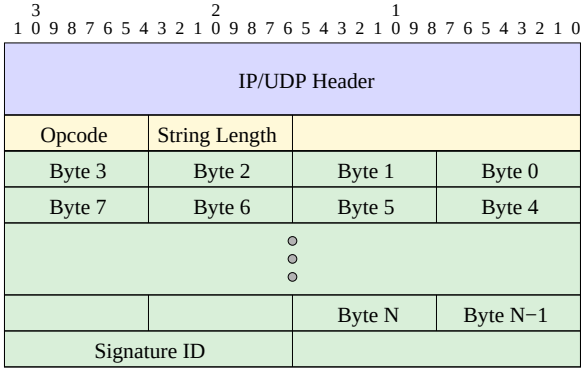

Figure A.1: Analyzer control packet format for adding and deleting strings from the analyzer (opcodes x70 and x72).

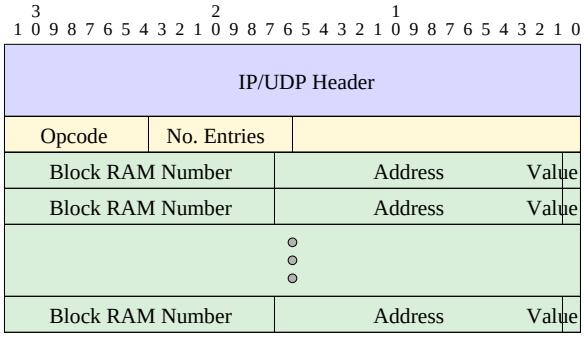

Figure A.2: Bloom filter control packet format (opcode x74).

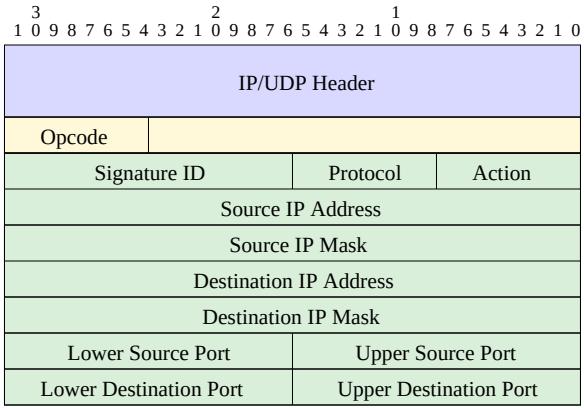

Figure A.3: Header entry control packet format (opcode x76).

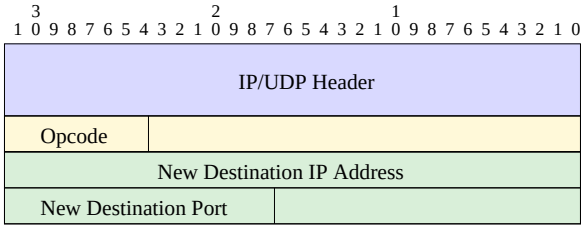

Figure A.4: Change of destination address for alert reporting (opcode x80).

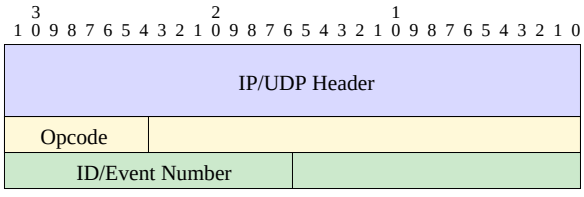

Figure A.5: Read status/event control packet format (opcodes x78, x82 and x84).

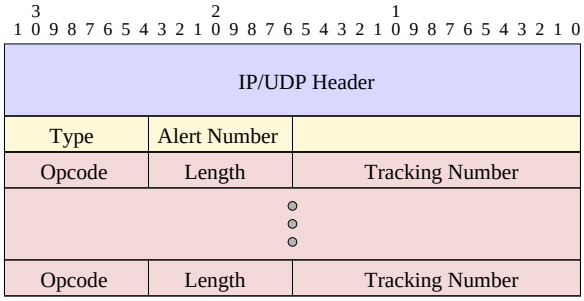

Figure A.6: Type 1 alerts acknowledge the receipt of a control opcode.

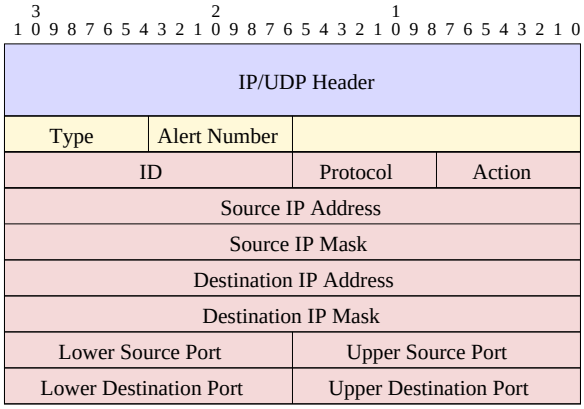

Figure A.7: Type 2 alerts contain the header entry that was read.

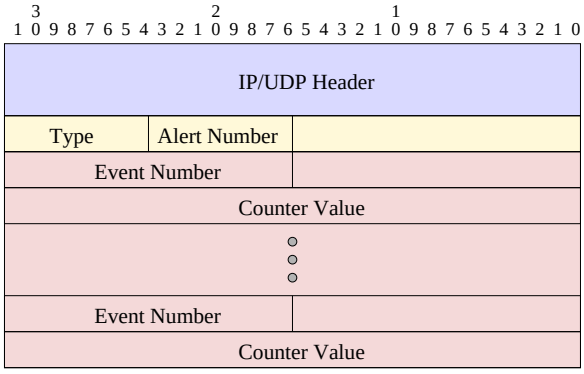

Figure A.8: Type 3 alerts are used to return a statistic.

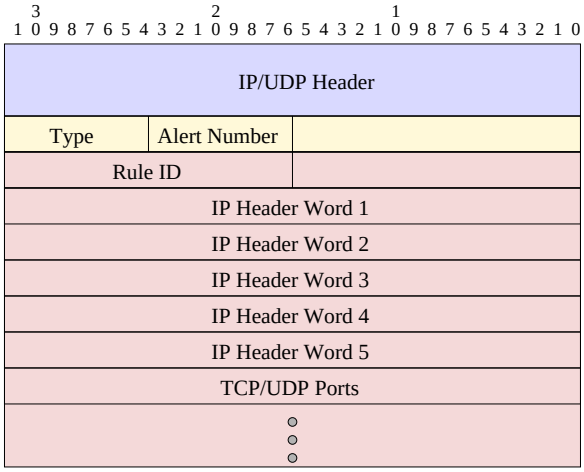

Figure A.9: Type 4 alerts are used to inform a software process that a rule has matched. Only the packet header is returned.

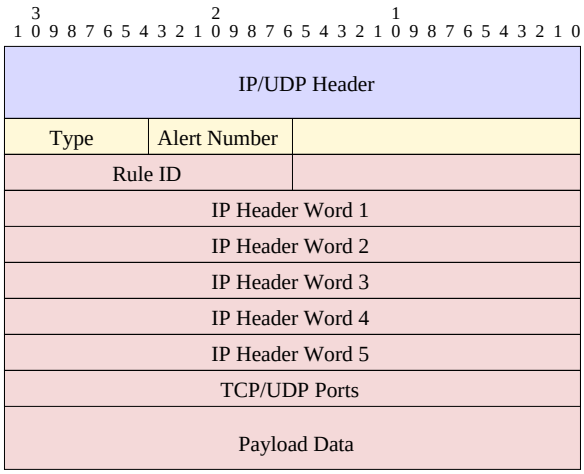

Figure A.10: Type 8 alerts are used to inform a software process that a rule has matched. The entire packet is returned.

A.2 SIFT

The software used for the SIFT system differs from that used in Snort Lite in several fundamental ways.

1) The signature deletion feature was included.

2) The ability to add/delete header rules was removed.

3) All communication is performed via the communication wrapper.

4) Internal file formats are abstracted away from the user.

The application can be executed with the following command:

```
$ java ContentFpga [-s Coefficients File] [-n Minimum String
Length] [-m Maximum String Length] [-v Verbose Level] [-o
Simulation Output File] [-r Rule File]
```

The -s option specifies where to find the random coefficients VHDL file used by the Bloom engines. The -n option specifies the minimum string length programmable in the hardware. The -m option specifies the maximum string length that can be searched by the hardware. The -v option specifies the amount of output printed to the screen while operating the system. A value of 0 means very little may be output, while a value of 2 means that much will be output. The -o option specifies where to write a simulation file of all operations sent to the hardware. Finally, the -r options specifies an initial rule file to load into the system. The -s, -n, and -m options are required. All others are optional.

While the system is operating, the following commands can be used: *file*, *rf*, *add*, *remove*, *dump*, and *exit*. The *file* command takes as a parameter the name of a rule file to load into the system. The *rf* command takes as a parameter a rule file to remove from the hardware. The *add* command is used to manually enter a single rule. The *remove* command is used to manually remove a rule. The *dump* command is used to view the contents of a specific Bloom filter engine. Finally, the *exit* command closes the program.

The format of rules is as shown below. Rules must include a signature, an action, a value to change the ToS field to upon a signature match, and finally the signature. ID numbers can range from 1 to 65535. There are four possible actions: *alert*, *filter*, *return*, and *filterreturn*. The *alert* action informs the software process

what signature was found. The *filter* action informs software of what signature was found and drops the packet from the network. The *return* action informs software what signature matched and also returns the entire packet that caused the match. Finally, the *filterreturn* action returns the signature ID of the matching signature and the offending packet while dropping the packet from the network.

ID	action	ToS	Signature
10	alert	0	Washington
20	filter	0	University
30	return	15	|00 00 01 1c ab ff|hex
40	filterreturn	31	ViRu$

When the application loads, several events occur. First, the coefficient file for the Bloom filter hash functions is read into memory. Second, space is allocated for the various Bloom filters that are in use by the hardware. Third, the system attempts to open a communication channel to the hardware. This is performed by sending out a "hello" packet to the hardware. The system waits a timeout period for a response. If no response arrives, the application closes. If a response returns, the system is ready to begin. Once the communication channel is established, the system waits for user input.

At this point, the user can specify a file of rules to add or manually enter them one by one. When a rule is sent to the hardware, there are three possible outcomes. First, the user will be greeted with the response *Success...* which means that the rule was successfully added. Second, the user may be informed that no response was received within a timeout period. In this case, communication to the hardware has been lost or there has been an error. Finally, the user may be told that the signature already exists in the hardware. In this case, no control messages will be sent to the hardware.

The SIFT software implements two opcodes. The first (x1), correlates a signature and an action in SRAM. When a match occurs, the signature that caused the match is hashed and looked up in SRAM to determine what action to take as well as what ToS value to use. The format for opcode x1 is shown in Figure A.11. The second opcode (x2) sets/resets bits in a Bloom filter and is shown in Figure A.12.

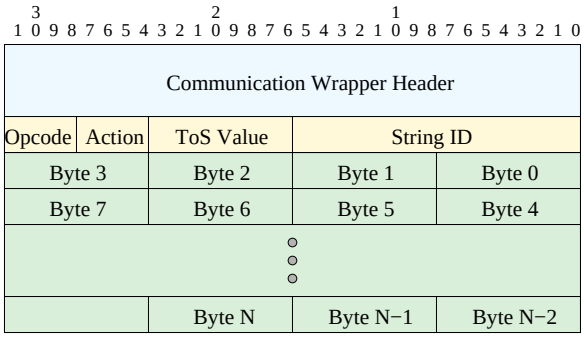

Figure A.11: SIFT opcode 1 is used to correlate signatures with actions.

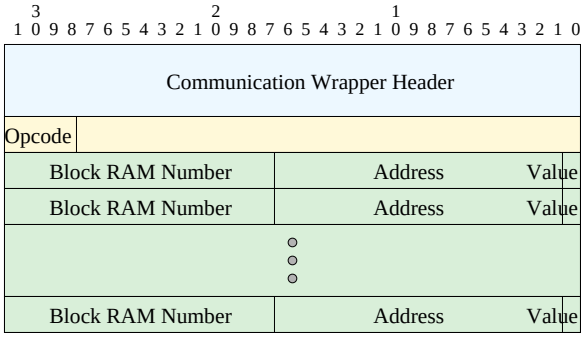

Figure A.12: SIFT opcode 2 is used to set/reset bits in a Bloom filter.

A.2.1 Software Bloom Filter

In order to remove signatures once they have been added to a Bloom filter, a counting Bloom filter must be used, as described in [36]. The software maintains an exact representation of what the Bloom filters in hardware hold with one exception. While the hardware allocates one bit for each bucket in the bit vector used by the Bloom filter, the software allocates an `int`. When a signature is added, the software increments the location. If the value changes from 0 to 1, a control message has to be sent to hardware. When a deletion is requested, the software decrements the hashed locations. If any locations transition to 0, the software sends a control message to hardware to reset those particular bit positions.

A.2.2 Communication Wrapper

In order to properly format data that is destined to pass through the communication wrapper, a simulation application was developed that converts a data stream into a

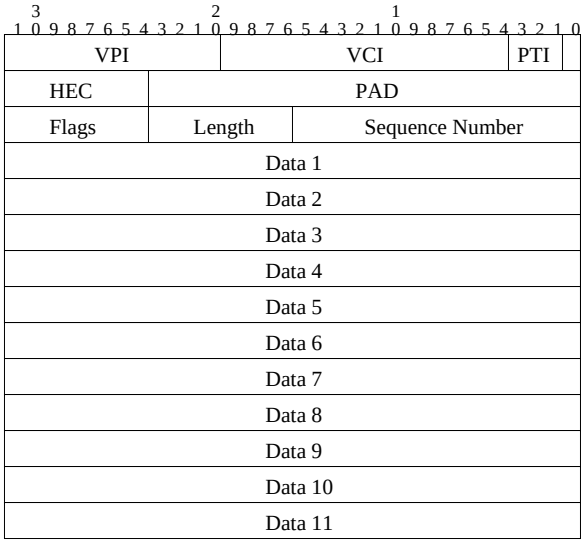

Figure A.13: The transport ATM cell consists of two words for ATM routing, one word of communication header, and eleven words of data.

properly formatted sequence of ATM cells that can be interpreted by the communication wrapper's inbound interface. The ATM cell format is shown in Figure A.13.

The internal format used by the communication wrapper consists of a 14 word ATM cell. The first five bytes are the standard ATM cell header with VPI, VCI, and HEC. The second word is padded out to word-align the rest of the data. The third word consists of the communication header, which has fields for flags, data length, and sequence number. There are three flags: acknowledgement, last cell, and version/hello. The acknowledgment bit (bit 31) is set by the wrapper when all data from a particular data stream has been received. The last cell bit (bit 30) is set when the current ATM cell represents the last cell of a data stream. The version/hello bit (bit 29) is set by software to send a query to the wrapper to see if it is operational. The wrapper provides the current version number of the application.

The 8-bit length field specifies how many bytes are valid in the subsequent payload of the cell. Bytes are aligned from left to right, with the left byte being the most significant. Valid values for the length are 1 byte to 44 bytes.

The 16-bit sequence number field orders data as it arrives. For example, sequence number 0 specifies that this is the beginning of the data flow and the bytes in the payload are the first bytes of this new flow. The wrapper currently supports sequence numbers in the range 0 to 63, allowing the maximum size of a data flow to be 2816 bytes.

The *CwTest* application takes a *.frm* file and converts it to a *.DAT* file that is readable by the FPX testbench, as shown below. To execute the application, use the following command:

```
$ java CwTest <Input Frame File> <Output DAT File>
```

new_frame	new_cell
20000000	00000250
200001fb	C8000000
200008b7	40240000
20021e9f	20000000
20020d09	200001fb
200413d7	200008b7
20040d05	20021e9f
20061b93	20020d09
20061115	200413d7
	20040d05
	20061b93
	20061115
	00000000
	00000000

A.3 Rule Processor

The software for the rule processor is more complicated than the other two applications due to the memory management. The command line arguments are the same as that used by the SIFT controller and are included here again for clarity.

```
$ java RuleFpga [-s Coefficients File] [-n Minimum String Length]
[-m Maximum String Length] [-v Verbose Level] [-o Simulation
Output File] [-r Rule File]
```

However, rules take on a slightly different form for this system. A header is specified and multiple signatures are allowed. The application dynamically assigns a header ID to any headers and signature IDs to any signatures found in the rule and loads the rule processor.

A header rule is constructed from the protocol, source IP address, source port, destination IP address, destination port, and various options that can appear. This value is hashed to determine whether it has already been identified. If it has, the old ID value is used. Otherwise, a new ID is given to it. Once the ID has been determined, the rule ID to header ID mapping in SRAM is updated in the rule processor.

A similar tactic is utilized for keeping track of signatures. Once all the signatures in the rule have been numbered, the position in SRAM is determined to place the mappings for signature ID to rule ID. If this is the first occurrence of the signature, the signature ID is used as a direct index to write SRAM. This is updated in the hardware and software. If this is not the first occurrence of the signature, the linked list starting at the direct index of the signature ID in SRAM is followed. Here, two SRAM updates will be written. The next pointer for the last entry will be modified, and the new rule ID will be appended to the end of the list. The number of signatures in the rule is counted, and the value is loaded into the block RAM vector in hardware. Rule IDs are parsed from the *SID* field in Snort rules.

The control formats for writing SRAM are shown in Figure A.14. The same format is used to write both banks of SRAM. However, bank 0 is specified by using opcode x1, and bank 1 is specified by using opcode x2. Multiple writes to a bank of SRAM can be given in a control message. In order to read SRAM, the format of Figure A.15 is used. Again, the same format is used to read from both banks of SRAM. Bank 0 is read by using opcode x3, and bank 1 is read by using opcode x4. Only one read can be specified per message. In order to write BRAM coefficients, the format of Figure A.16 is used. Multiple coefficients can be specified in a given message. The default VCI for control traffic is x25 (37_{10}). Match traffic defaults to VCI x2d (45_{10}).

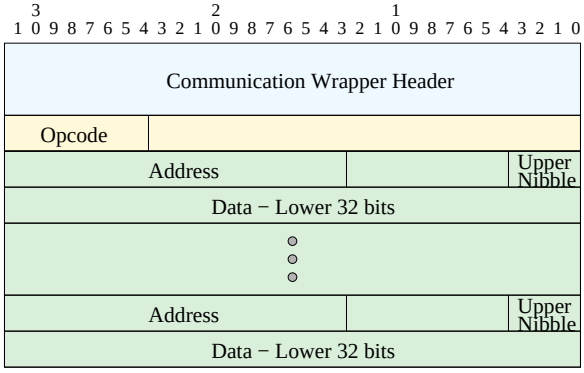

Figure A.14: SRAM writes take this form. The address is 19 bits. The upper nibble of the data is placed in the same word as the address. The remaining 32 bits of data are placed in the subsequent line. Opcode x1 writes SRAM bank 0, which contains the linked lists for string ID to rule ID mappings. Opcode x2 writes to SRAM bank 1, which holds the rule ID to header ID mapping.

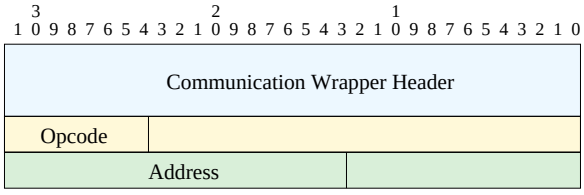

Figure A.15: A SRAM read consists of supplying a single address. Only one read can be performed at a time. Opcode x3 reads from SRAM bank 0, and opcode x4 reads from bank 1.

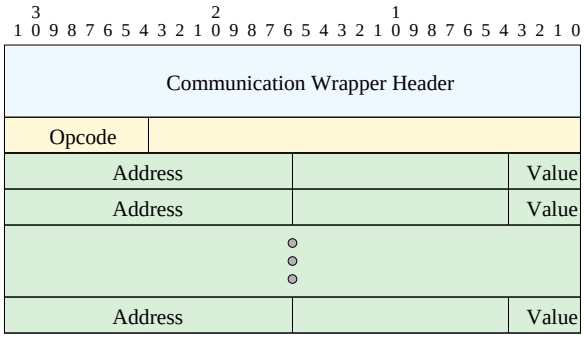

Figure A.16: This is the format of opcode x5. To load a BRAM coefficient, a 16-bit address is supplied along with the 4-bit value to write.

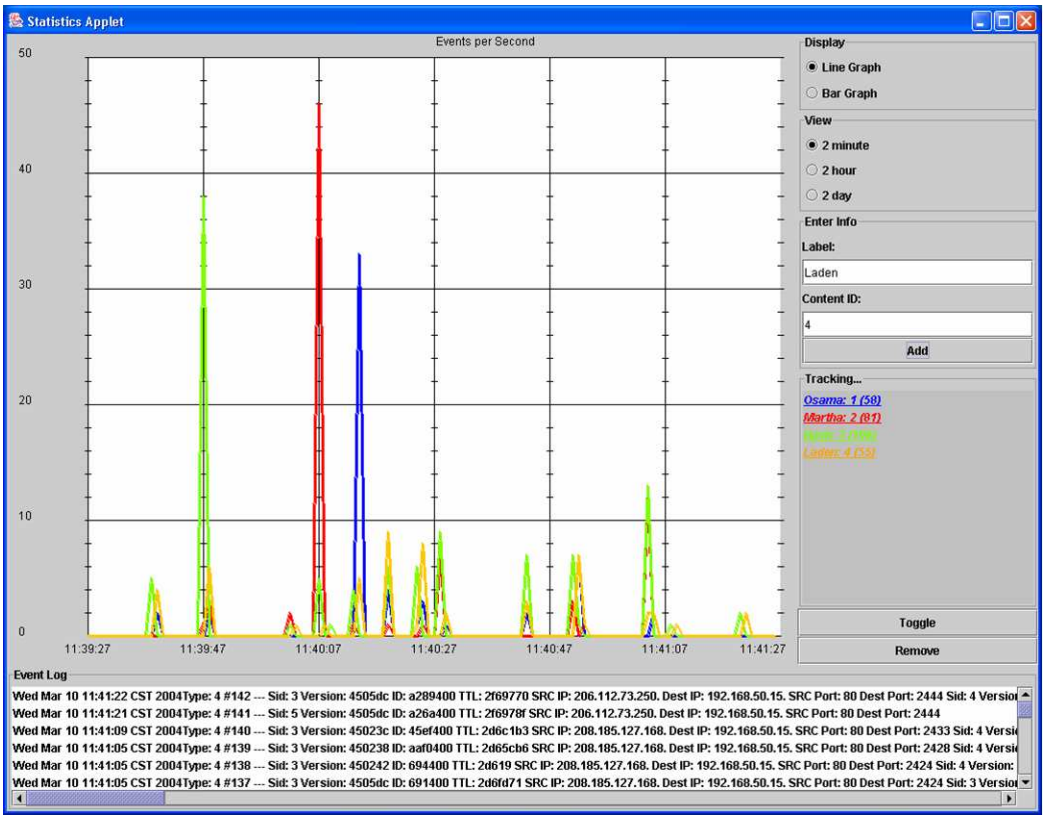

Figure A.17: The statistics graphing application in line graph mode.

A.4 Statistics Graphing

The statistics graphing application was developed in order to view the number of string matches that occur per second. The application receives UDP alert messages from hardware applications and displays how many occurrences of a specified event have occurred. The graph has two modes of display. The first is line graph mode, as shown in Figure A.17. The y-axis auto-scales to reflect changes in the number of events that have occurred per second. An unlimited number of events could be monitored simultaneously, but beyond ten the graph becomes too cluttered to interpret.

A second mode of display is also available.In this mode, up to ten bar graphs can be shown that reflect the current event count in relative scale.

To run the application, use the command:

```
$ java StatApp <Port#>
```

The port number specified tells the application what port to listen on for alert messages from a hardware circuit.

There are four areas of note in the display window. First, the main component is the line/bar graph component. This is where the graph is displayed. Second, directly below the display is the event log. When alert messages arrive, a time stamp along and type of event are printed to the event log. The format for a alert message event is *Current Time – Event ID – Source IP – Destination IP – Ports*. Third, the control area of the application is on the right side. In here can be found the ability to change the display from line graph to bar graph, the ability to change view mode (line graph only) from two-minute, two-hour, or two-day mode, and the ability to enter information for a new event to monitor. Fourth, directly below control settings is the list of current alerts being tracked. This list shows the color, the label entered (if any), the alert number, and the number of total alerts so far (found in parenthesis). An alert can be removed by selecting it and clicking on the remove button. The graphs that are displayed can be toggled on and off by hitting the toggle button.

Appendix B

Laboratory Configuration

B.1 Configuration of Snort Lite

Snort Lite operated in active mode between a cluster of computers and the external Internet, as shown in Figure B.1. A control host from the internal network programmed the device with rules. The control host also acted as the receiver of alerts, which were displayed in real-time. The system was placed in the GVS-1500.

Interfaces

Snort Lite was designed to use the line card interface of the RAD. Data traffic was scanned by the device as it travels in both directions. That is, traffic from the cluster of computers to the Internet was scanned, and traffic from the Internet to the cluster of computers was scanned. In order to route the appropriate traffic to the device, the following NCHARGE commands were issued:

```
c snortlite.bit # load the Snort Lite bitfile into the RAD
t 0 33 3 0 0 1 # process traffic from the Internet
t 80 33 3 0 0 0 # process traffic from the Intranet
t 0 32 1 0 0 1 # send control traffic to the software controller
```

The first command loads the Snort Lite bitfile into the RAD to begin processing. The second command routes Internet traffic coming from the system backplane into the RAD line card interface. The second command routes Intranet traffic coming from the Gigabit line card interface into the RAD line card interface. Intranet traffic is distinguished from Internet traffic by the VPI routing number [107]. The flow of data through the processing stack is shown in Figure B.2.

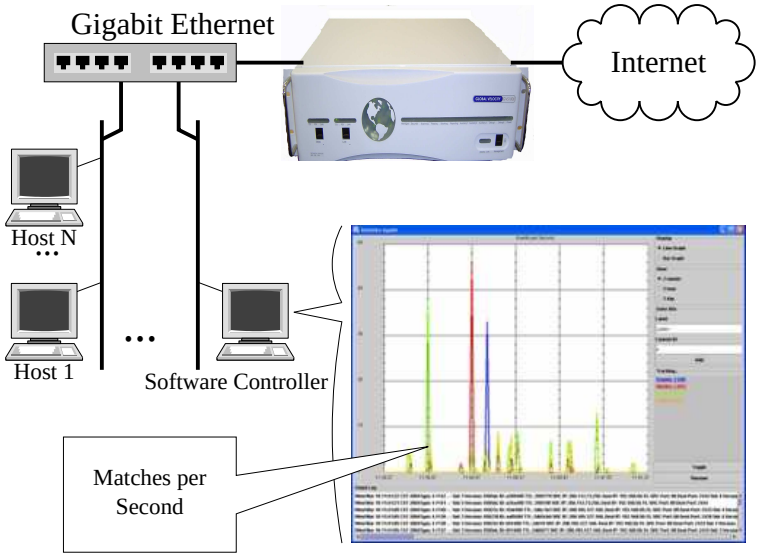

Figure B.1: To test the system with live traffic, the rule processor was placed between a host network and the Internet. Traffic was scanned in both directions. Summaries of the activity are sent to a software controller, which in turn plotted rule matches per second.

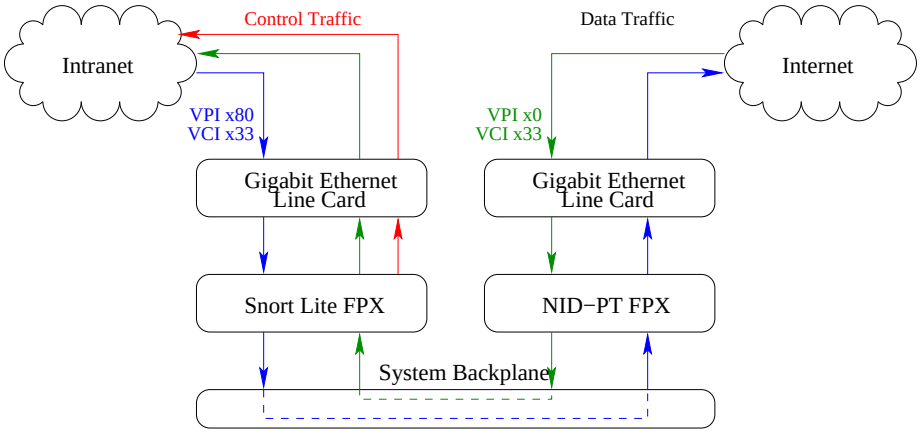

Figure B.2: Snort Lite used two FPX cards and two Gigabit Ethernet line cards. Data traffic from the Internet is shown in green, and data traffic from the Intranet is shown in blue. Control traffic from Snort Lite is shown in red.

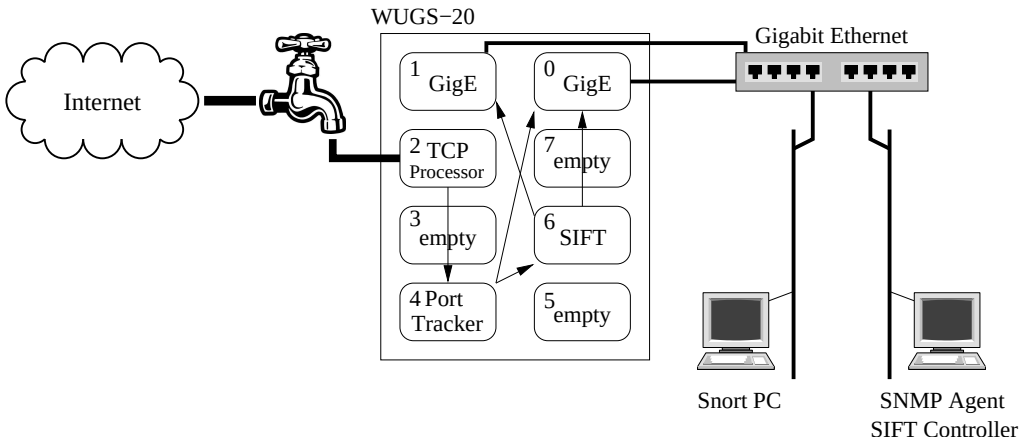

Figure B.3: The 8-port WUGS-20 was populated with five devices: three FPX cards and two Gigabit Ethernet line cards. One Gigabit Ethernet line card transmitted unfiltered data to a Snort PC, and the other line card transmitted control information.

B.2 Configuration of SIFT

The SIFT architecture used the WUGS-20 environment to process data, as shown in Figure B.3. For this application, a mirror of all network traffic used at Washington University was available. The mirrored traffic was sent to the TCP processor, where it was analyzed and flows were reconstructed. From here, it was encoded and sent through the switch fabric to the port tracker application [96]. Finally, the encoded traffic is sent to the SIFT circuit where signature matching and header rule processing are performed.

There were two scripts to configure this system. The first programmed the NID routes on each of the FPX cards. The commands were sent to a separate NCHARGE process for each FPX card.

```
./basic_send 6.0 c sift.bit # load the SIFT bitfile
./basic_send 2.0 c streamextract.bit # load the TCP Processor
./basic_send 4.0 c porttracker.bit # load the Port Tracker bitfile
# Program TCP Processor NID routes
./basic_send 2.0 t 0 32 2 0 0 0
./basic_send 2.0 t 0 33 0 2 0 0
./basic_send 2.0 t 0 23 2 0 0 0
./basic_send 2.0 t 0 34 3 0 0 0
./basic_send 2.0 t 0 37 0 0 0 0
```

```
# Program Port Tracker NID routes
./basic_send 4.0 t 0 32 2 0 0 0
./basic_send 4.0 t 0 34 3 0 0 0
# Program SIFT NID routes
./basic_send 6.0 t 0 25 3 0 0 0
./basic_send 6.0 t 0 34 2 0 1 0
./basic_send 6.0 t 0 33 1 0 0 0
./basic_send 6.0 t 0 32 1 0 0 0
```

The second script programmed the unidirectional switch routes from one switch port to another. Routes specify the source port and VCI and the destination port and VCI.

```
unidir(6,50,0,50) # Control Traffic
unidir(4,50,0,50)
unidir(2,55,0,50)
unidir(2,50,0,50)
unidir(6,51,1,51) # Data Traffic
unidir(4,52,6,52) # Encoded Data Traffic
unidir(2,52,4,52)
unidir(1,51,2,50) # SIFT Control Traffic
unidir(1,37,6,37)
unidir(6,37,1,37)
```

B.3 Configuration of Rule Processor

The rule processing framework leveraged the WUGS-20 for configuration, as shown in Figure B.4. The four FPX cards used by the system instantiated a TCP Processor, header processor, payload scanner, and rule processor. The header processor and payload scanner communicated with the rule processor. The device was configured to be passive. The Gigabit Ethernet line card was between the devices and a software controller that added header rules to the header processor, signatures to the payload scanner, and rule specifications to the rule processor. Alert messages were sent from the rule processor out via the Gigabit Ethernet line card to a monitor PC.

Again, there were two scripts that needed to be run to configure the system. The first configured the FPX cards and is shown below.

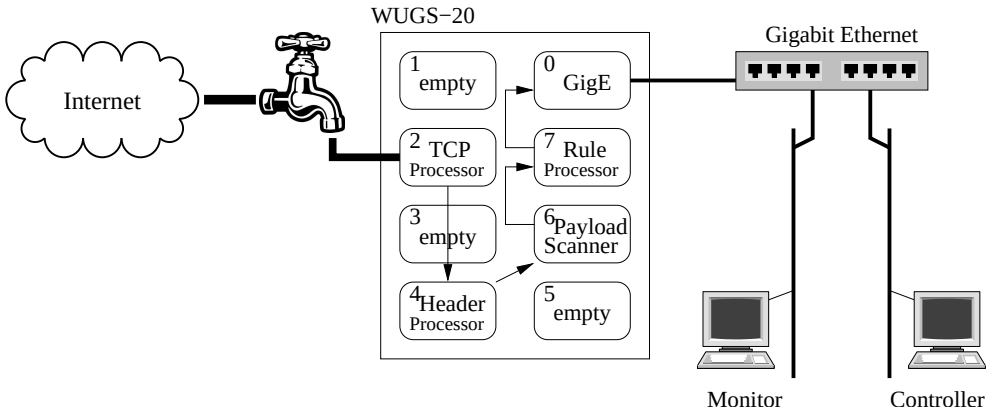

Figure B.4: The 8-port WUGS-20 was populated with five devices: four FPX cards and one Gigabit Ethernet line card.

```
./basic_send 2.0 c streamextract.bit # load the TCP Processor
./basic_send 4.0 c headerprocessor.bit # load the Header Processor
./basic_send 6.0 c payloadscanner.bit # load the Payload Scanner
./basic_send 7.0 c ruleprocessor.bit # load the Rule Processor
# Program TCP Processor NID routes
./basic_send 2.0 t 0 32 2 0 0 0
./basic_send 2.0 t 0 33 0 2 0 0
./basic_send 2.0 t 0 23 2 0 0 0
./basic_send 2.0 t 0 34 3 0 0 0
./basic_send 2.0 t 0 37 0 0 0 0
# Program Header Processor routes
./basic_send 4.0 t 0 21 2 0 0 0 # Control
./basic_send 4.0 t 0 34 3 0 0 0 # Packet Data
./basic_send 4.0 t 0 2a 1 0 0 0 # Match IDs
# Program Payload Scanner routes
./basic_send 6.0 t 0 23 3 0 0 0 # Control
./basic_send 6.0 t 0 34 2 0 0 0 # Packet Data
./basic_send 6.0 t 0 2a 1 0 0 0 # Match IDs
# Program Rule Processor routes
./basic_send 7.0 t 0 25 3 0 0 0 # Control
./basic_send 7.0 t 0 2a 2 0 0 0 # Match IDs
```

Finally, the switch routes were configured, as shown below.

```
unidir(6,50,0,50) # Control Traffic
unidir(4,50,0,50)
unidir(2,55,0,50)
unidir(2,50,0,50)
unidir(4,52,6,52) # Encoded Data Traffic
unidir(2,52,4,52)
unidir(0,33,4,33) # Rule Control Traffic
unidir(0,35,6,35)
unidir(0,37,7,37)
unidir(7,37,0,37)
```

Appendix C

Source Files

C.1 Common Directory Structure

A common directory structure for all projects was used, as shown in Figure C.1. The *backend* directory holds the Xilinx build files, such as *edf*, *edn*, *bit*, and constraint files. The *doc* directory holds all the documentation created for the project like figures and presentations. The *sim* directory contains all files related to simulating the design, such as *DAT* files. The compiled VHDL source is held in a subdirectory called *work*. The *software* directory contains all the control software or data formatting scripts for the project. The *syn* directory holds all *Synplicity* project information. The synthesis directory is where the *edf* file is created. Finally, the *vhdl* directory holds all the VHDL source files for the project.

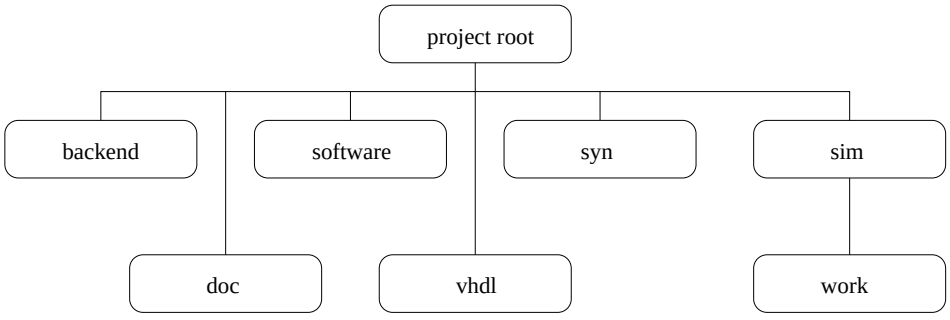

Figure C.1: Each project has its own group of six folders: backend, doc, sim, software, syn, and vhdl.

C.2 Scripts

Makefiles are found in the *backend* and *sim* directories to ease the development process. From the project root, use the following commands to simulate a design:

```
$ cd sim
$ make compile > log.txt
$ make sim
```

The `compile` command compiles all VHDL source files that are listed in the makefile, placing the compiled source files into the *work* directory. The compilation output is placed in a log file to be examined if needed. (There will generally be a lot of output.) The `sim` command opens *Modelsim* to begin simulating the design.

From the project root, use the following commands to build a design:

```
$ cd backend
$ make build > log.txt
```

The `build` command performs several operations. First, it runs *Synplicity* to create the *edf* file for the project and moves it to the backend directory. Next the script runs the Xilinx *ngdbuild*, *map*, *par*, *trce*, and *bitgen* commands.

C.3 Snort Lite

The root directory for Snort Lite can be found in the FPX CVS tree at:

```
/project/arl/fpx/mike/cvsroot/snortlite
```

In order to checkout the project files, use the command

```
cvs -d /project/arl/fpx/mike/cvsroot/ co snortlite
```

Provided with the Snort Lite project files are a couple of additional directories. The first is the *graphing* directory where the statistics graphing application source is found. The second is a *top* directory. The top directory contains source code for the IP wrappers [23], the SDRAM memory controller [39], the SRAM memory controller [119], and the design's top level description. The synthesis of the Snort Lite application was separated from the top level for use with CSE535 [13] in the Fall 2003.

C.4 SIFT

The root directory for SIFT can be found in the FPX CVS tree at:

```
/project/arl/fpx/mike/cvsroot/sift
```

In order to checkout the project files, use the command

```
cvs -d /project/arl/fpx/mike/cvsroot/ co sift
```

One extra folder is provided with the SIFT distribution. The *snmp* folder contains source and configuration files for using SNMP and MRTG for statistics gathering.

C.5 Rule Processor

The root directory for the rule processor can be found in the FPX CVS tree at:

```
/project/arl/fpx/mike/cvsroot/rule
```

In order to checkout the project files, use the command

```
cvs -d /project/arl/fpx/mike/cvsroot/ co rule
```

C.6 Communication Wrapper

The root directory for the rule processor can be found in the FPX CVS tree at:

```
/project/arl/fpx/cvsroot/SAIC/commwrapper
```

In order to checkout the project files, use the command

```
cvs -d /project/arl/fpx/cvsroot/ co SAIC/commwrapper
```

The communication wrapper's *backend* directory does not contain information for building the design, as the communication wrapper is not intended to be the top level. However, the necessary *edn* files have been included.

Appendix D

Additional Figures

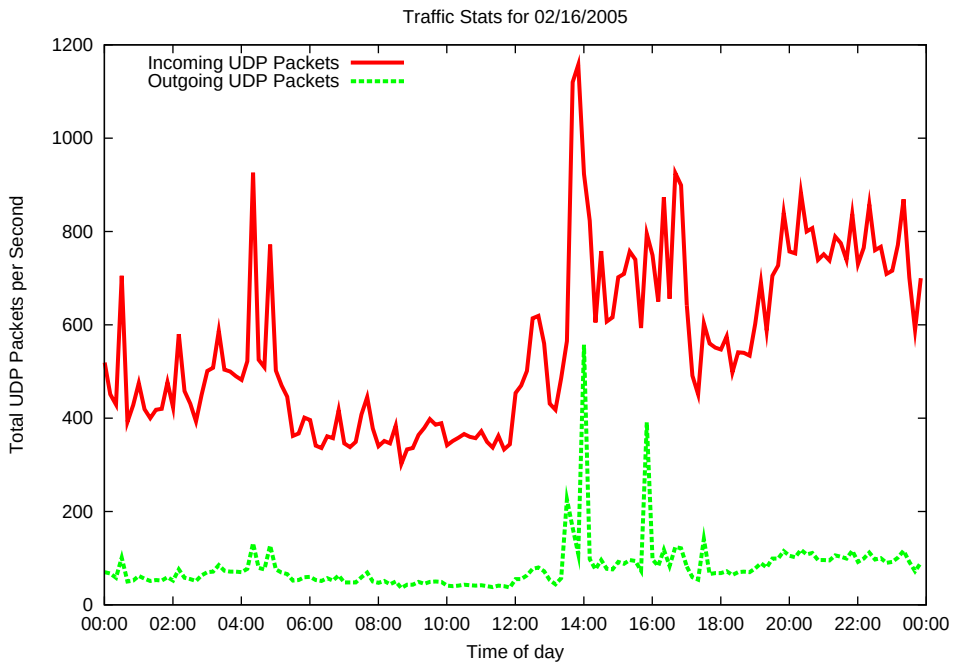

Figure D.1: The number of incoming and outgoing UDP packets versus time on Feb 16, 2005.

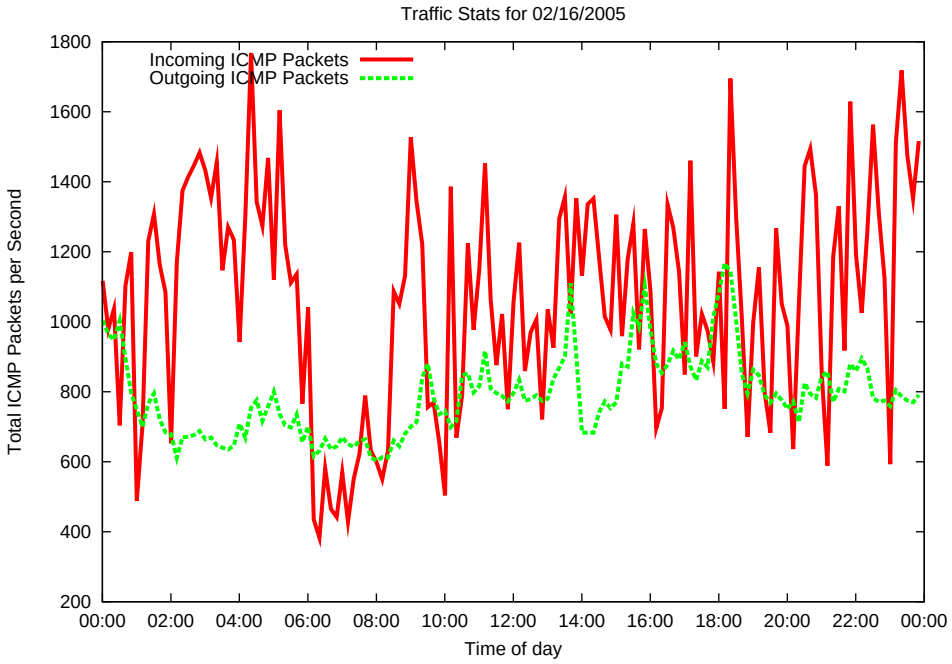

Figure D.2: The number of incoming and outgoing ICMP packets versus time on Feb 16, 2005.

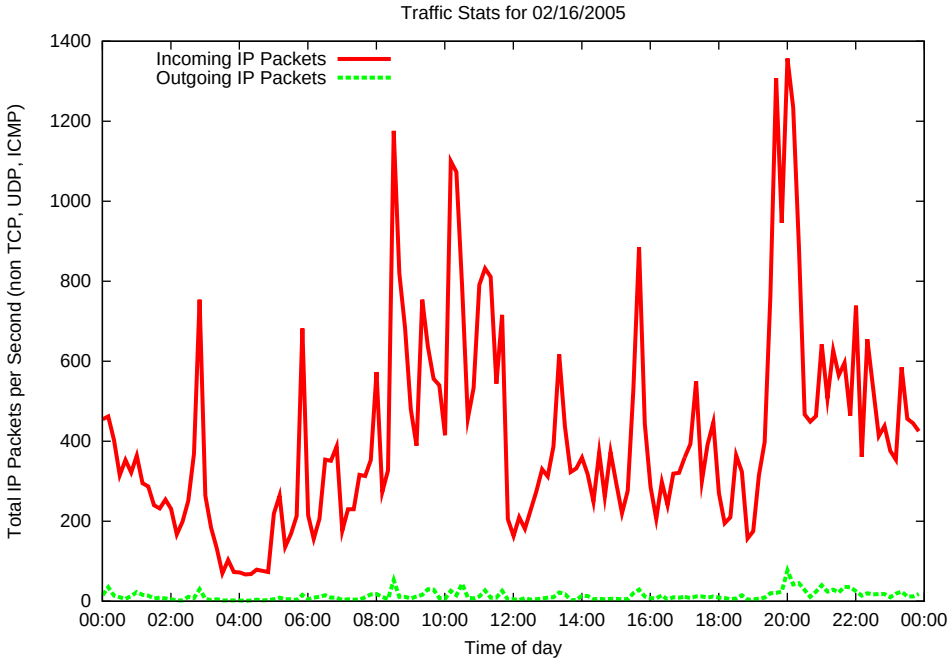

Figure D.3: The number of incoming and outgoing IP packets (not TCP, UDP, nor ICMP) versus time on Feb 16, 2005.

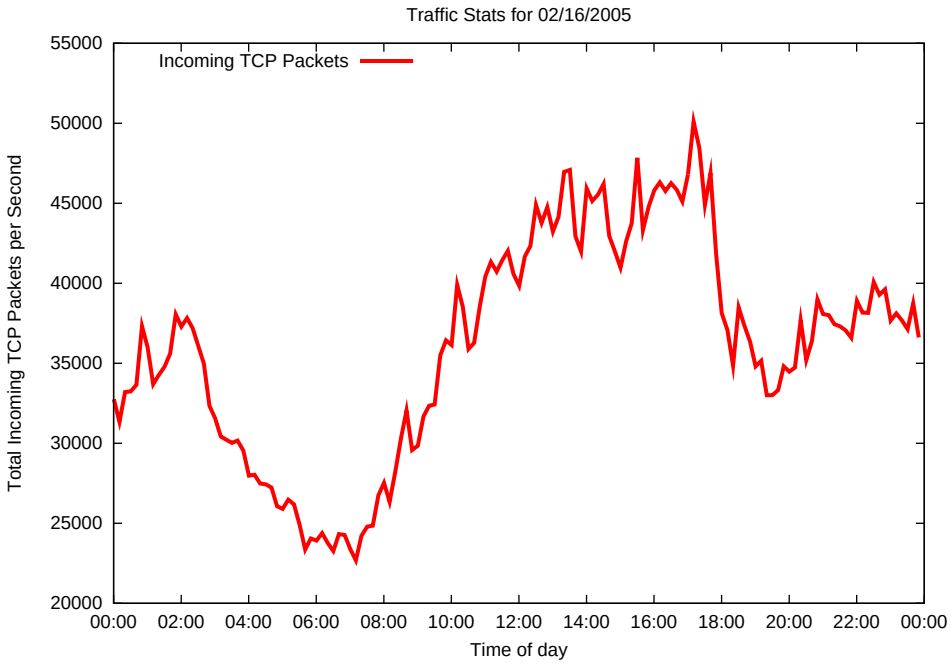

Figure D.4: The number of incoming TCP packets per second versus time on Feb 16, 2005.

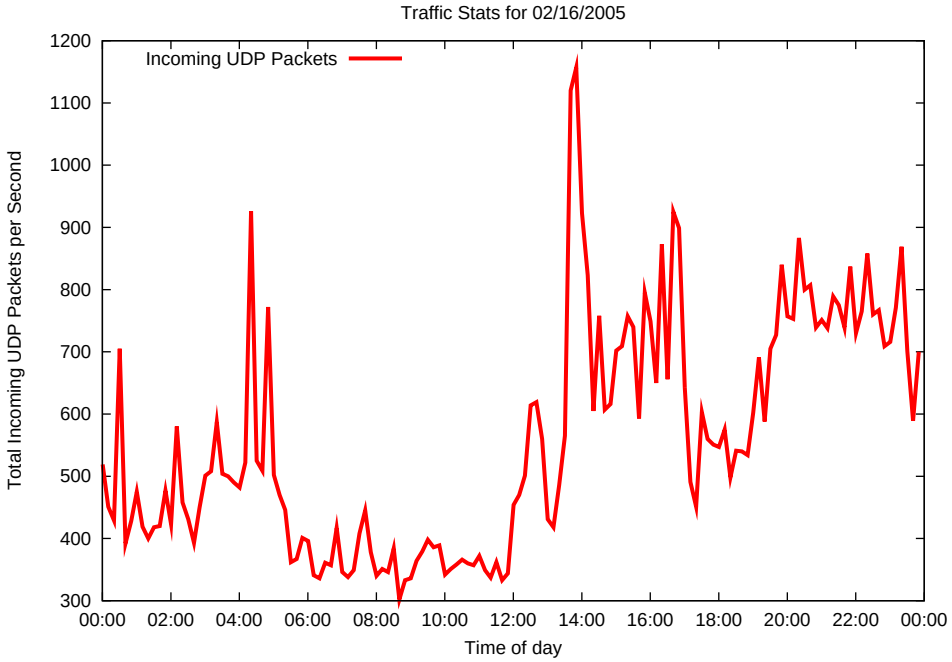

Figure D.5: The number of incoming UDP packets per second versus time on Feb 16, 2005.

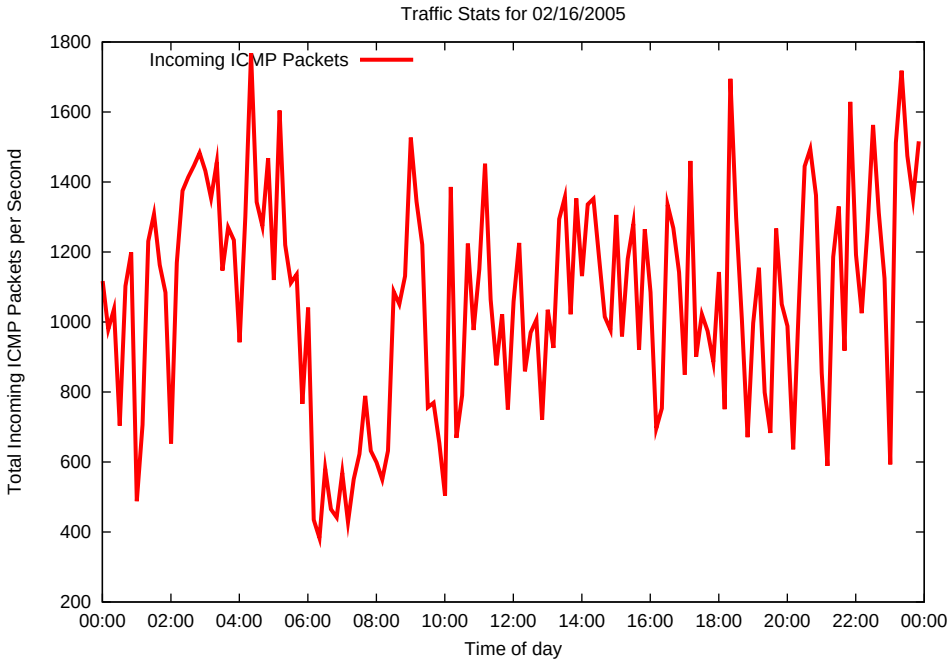

Figure D.6: The number of incoming ICMP packets per second versus time on Feb 16, 2005.

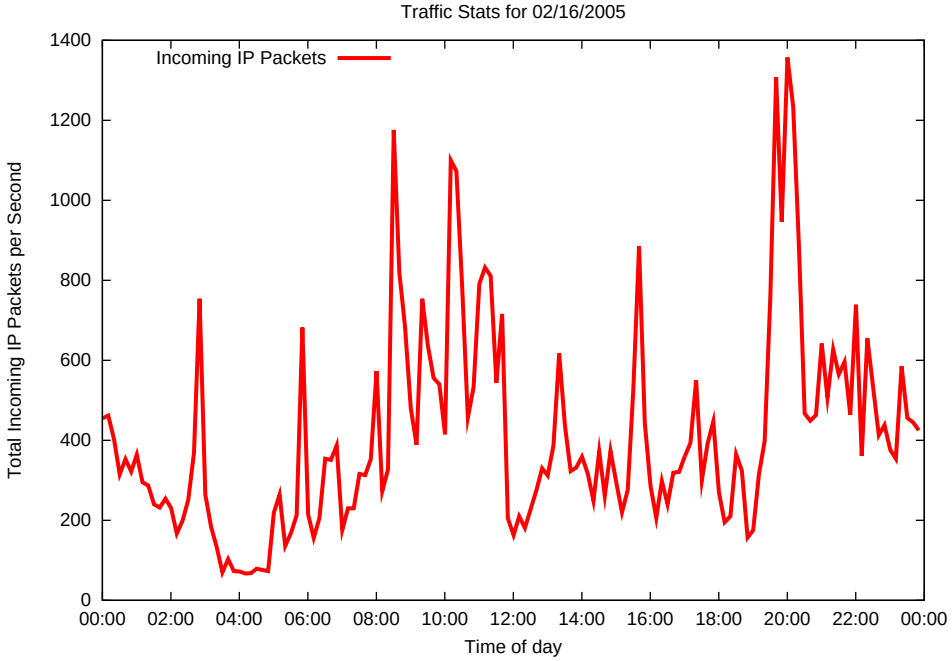

Figure D.7: The number of incoming IP packets per second versus time on Feb 16, 2005.

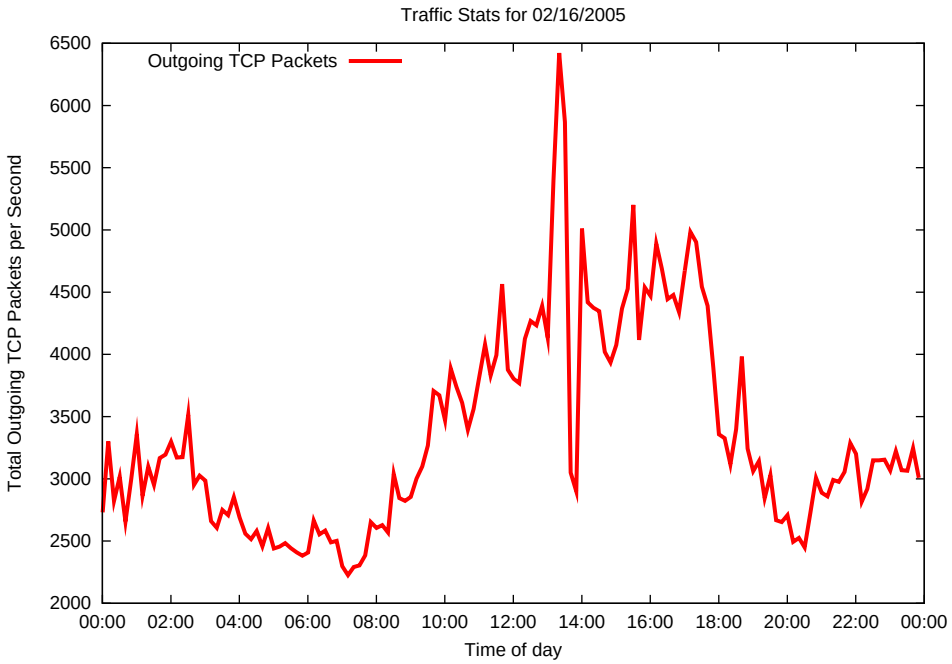

Figure D.8: The number of outgoing TCP packets per second versus time on Feb 16, 2005.

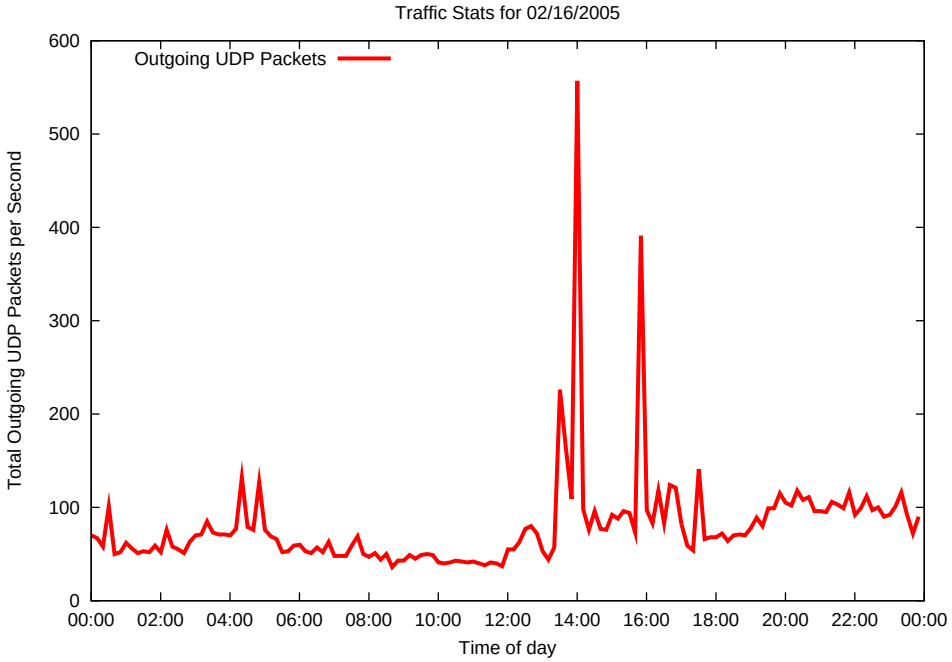

Figure D.9: The number of outgoing UDP packets per second versus time on Feb 16, 2005.

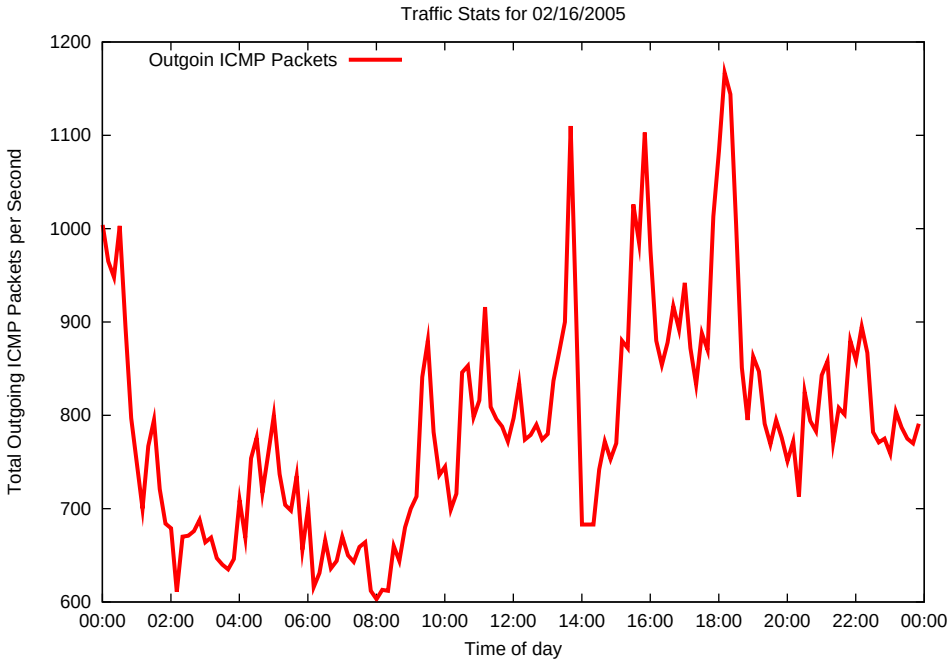

Figure D.10: The number of outgoing ICMP packets per second versus time on Feb 16, 2005.

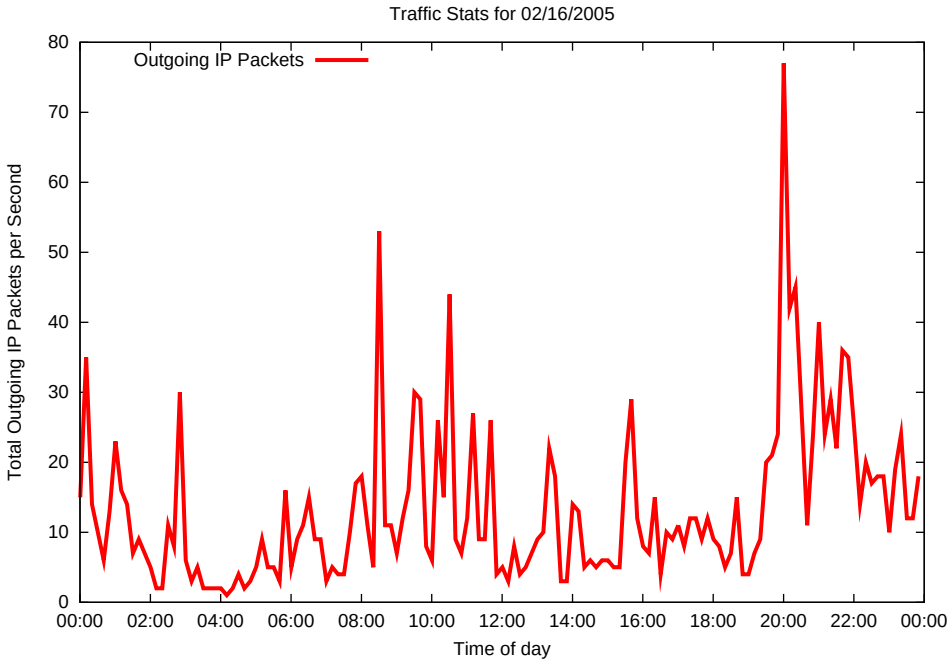

Figure D.11: The number of outgoing IP packets per second versus time on Feb 16, 2005.

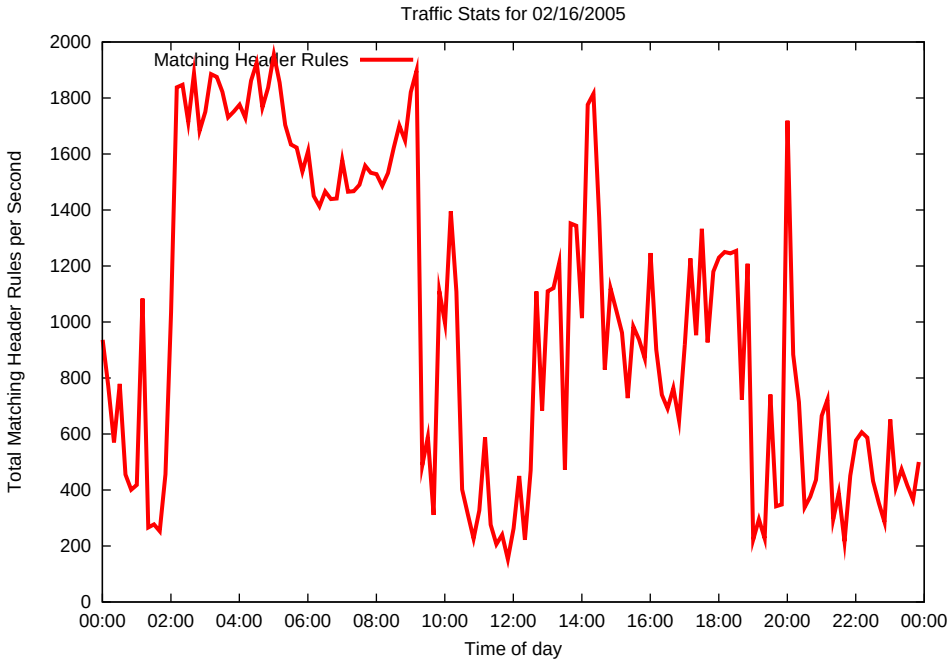

Figure D.12: The number of matching headers per second versus time on Feb 16, 2005.

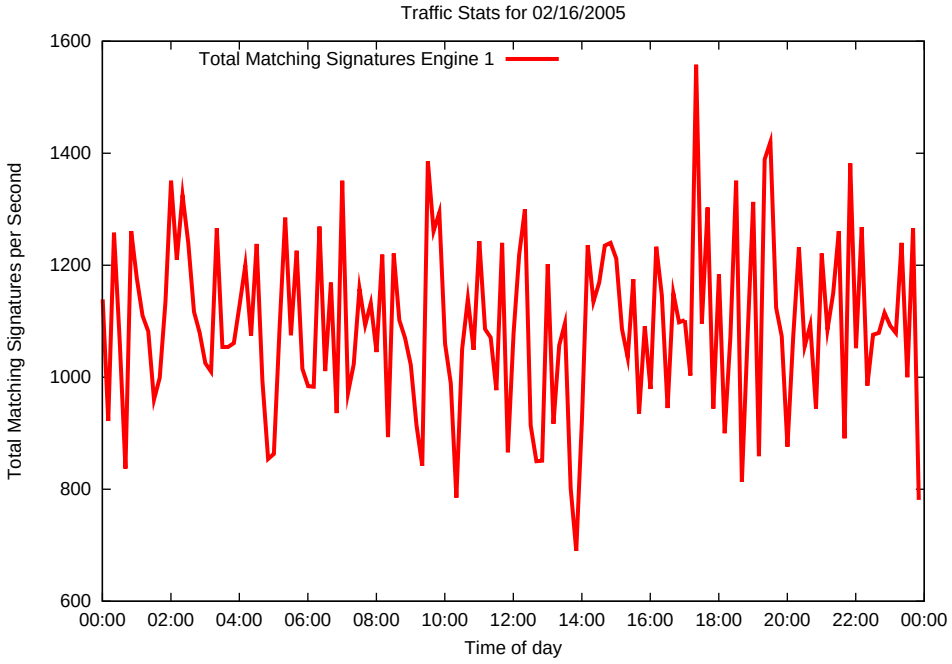

Figure D.13: The number of matching 4-byte signatures per second versus time on Feb 16, 2005.

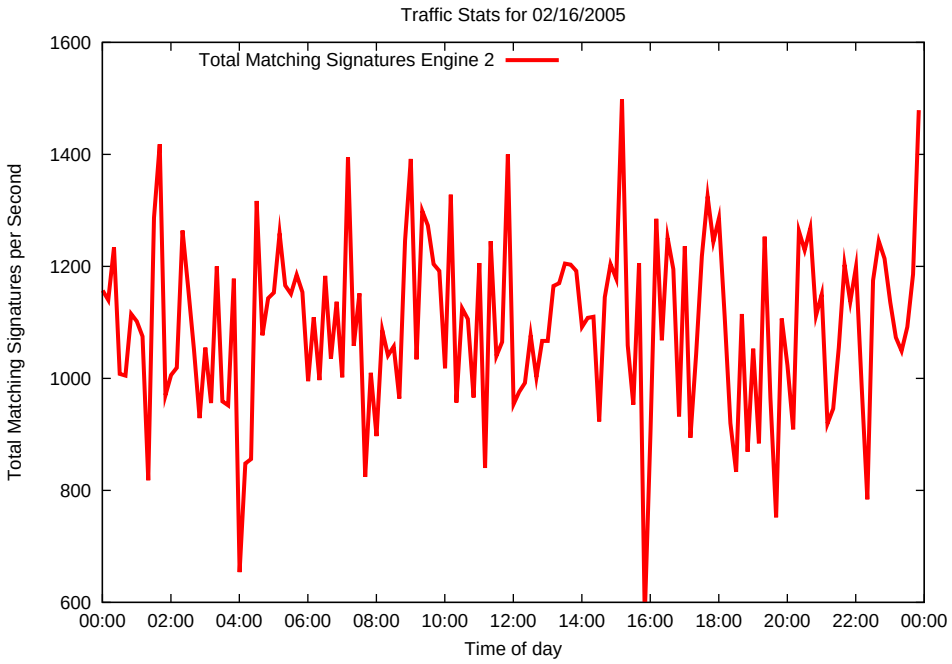

Figure D.14: The number of matching 6-byte signatures per second versus time on Feb 16, 2005.

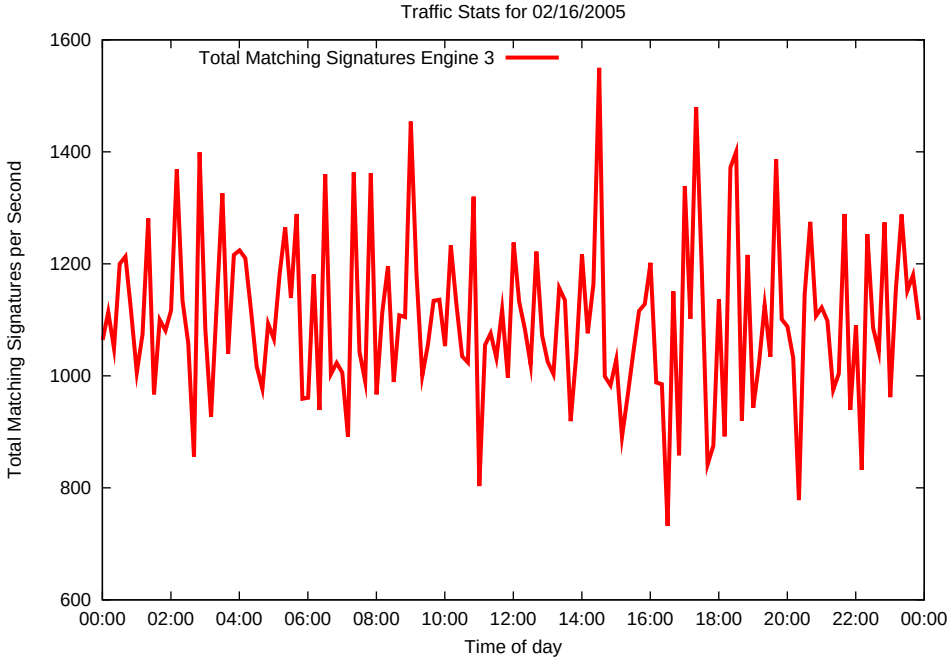

Figure D.15: The number of matching 8-byte signatures per second versus time on Feb 16, 2005.

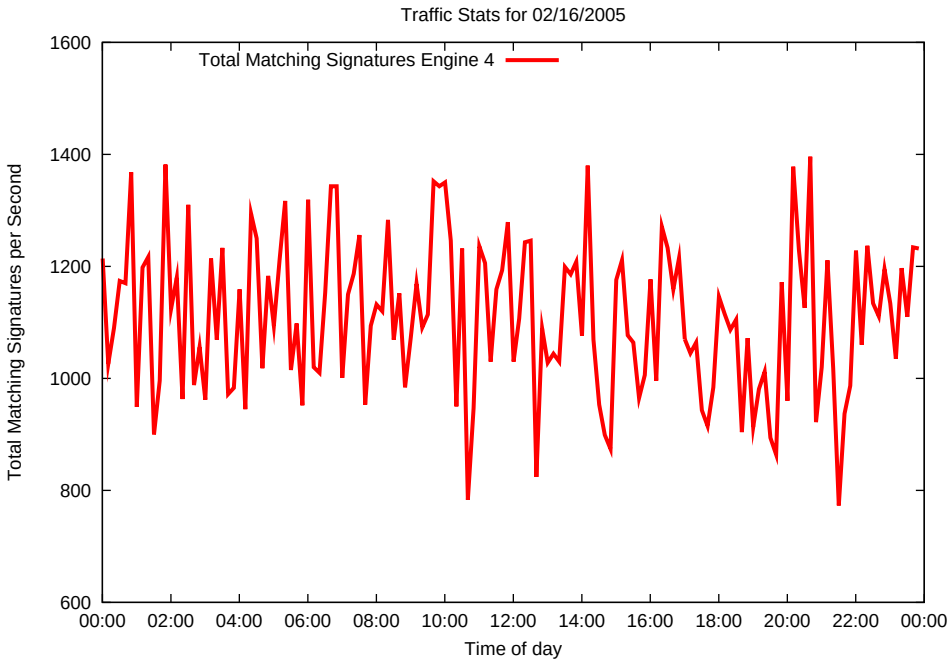

Figure D.16: The number of matching 10-byte signatures per second versus time on Feb 16, 2005.

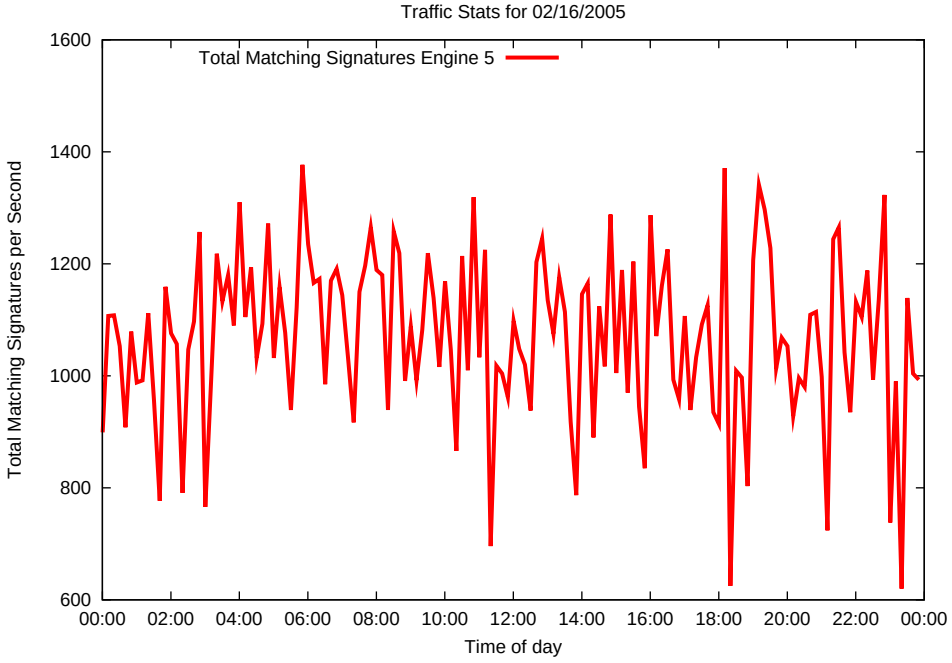

Figure D.17: The number of matching 12-byte signatures per second versus time on Feb 16, 2005.

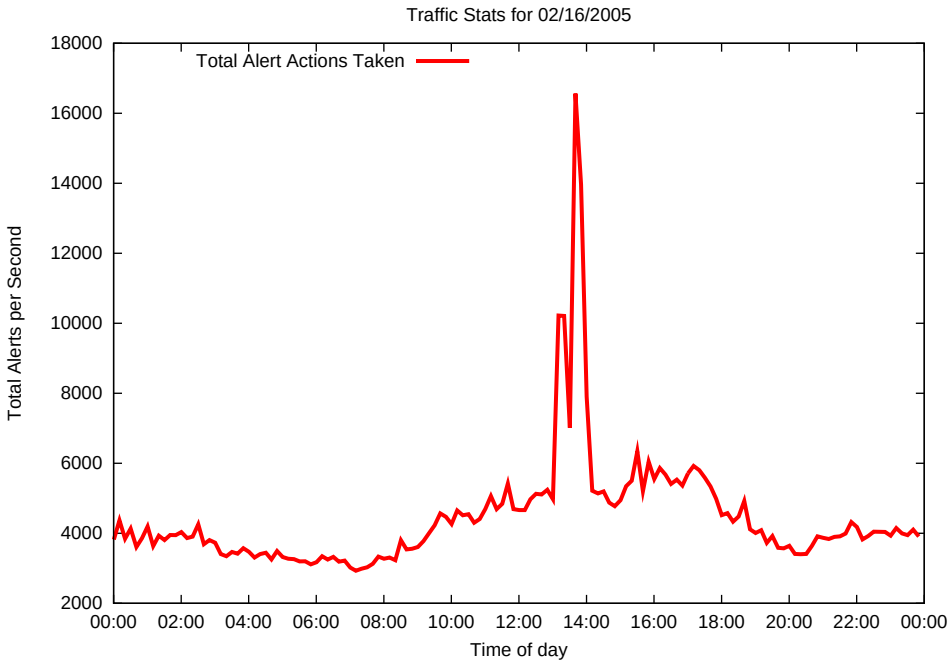

Figure D.18: The number of alert actions taken per second versus time on Feb 16, 2005.

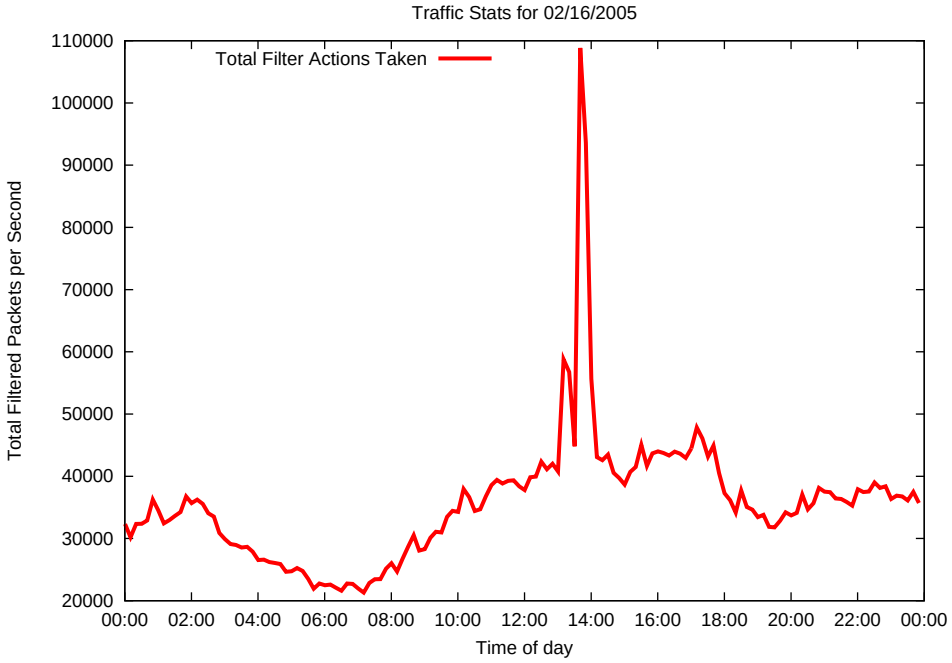

Figure D.19: The number of filter actions taken per second versus time on Feb 16, 2005.

Appendix E

List of Acronyms

AAL5	ATM Adaptation Layer 5
AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
BF	Bloom Filter
CAM	Content Addressable Memory
CLB	Configurable Logic Block
DDoS	Distributed Denial of Service
DFA	Deterministic Finite Automata
DoS	Denial of Service
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPX	Field programmable Port eXtender
FSM	Finite State Machine
GPP	General Purpose Processor
GUI	Graphical User Interface
HEC	Header Error Correct
ICMP	Internet Control Message Protocol
IDPS	Intrusion Detection and Prevention System
IDS	Intrusion Detection System
IP	Internet Protocol
IT	Information Technology
LAN	Local Area Network

LUT	Look-Up Table
malware	Malicious Software
MRTG	Multi Router Traffic Grapher
NFA	Non-deterministic Finite Automata
NIC	Network Interface Card
NID	Network Interface Device
NIDS	Network Intrusion Detection System
NRE	Non-Recurring Expense
OC	Optical Carrier
OSI	Open Systems Interconnect
OTN	Optional Tree Nodes
PBF	Partial Bloom Filter
PC	Personal Computer
QBF	Quad Bloom Filter
QoS	Quality of Service
RAD	Reprogrammable Application Device
RPF	Rule Processing Framework
RTN	Rule Tree Nodes
SBT	Suffix Based Traversing
SDRAM	Synchronous Dynamic Random Access Memory
SID	Signature ID
SIFT	Snort Intrusion Filter for TCP
SNMP	Simple Network Management Protocol
SRAM	Synchronous Random Access Memory
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
TOE	TCP Offload Engine
UDP	User Datagram Protocol
VCI	Virtual Circuit Identifier
VHDL	VHSIC Hardware Design Language
VHSIC	Very High Speed Integrated Circuit
VPI	Virtual Path Identifier
WAN	Wide Area Network
WUGS	Washington University Gigabit Switch
ZBT	Zero Bus Turnaround

References

[1] Digital attacks report - SIPS monthly intelligence description, economic damage - all attacks - yearly. Online as: `http://www.mi2g.net/cgi/mi2g/-sipsgraph.php` , September 2004.

[2] The radicati group. Online as: `http://www.radicati.com`, 2004.

[3] Snort Homepage. Online `http://www.snort.org`, 2004.

[4] Anti-phishing working group. Online as `http://www.antiphishing.org/`, January 2005.

[5] ARL home page. `http://www.arl.wustl.edu/`, January 2005.

[6] Global Velocity Web Page. `http://www.globalvelocity.com`, February 2005.

[7] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[8] Debra Anderson, Thane Frivold, Ann Tamaru, and Alfonso Valdes. Next-generation intrusion detection expert system (NIDES), Software Users Manual, Beta-Update release. Technical Report SRI–CSL–95–07, Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, May 1994.

[9] James P. Anderson. Computer security threat monitoring and surveillance. Technical Report Contract 79F26400, Box 42, Fort Washington, PA 19034, USA, 1980.

[10] Spyros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. Generating realistic workloads for network intrusion detection systems. In *WOSP'04: Proceedings of the Fourth International Workshop on Software and Performance*, pages 207–215. ACM Press, 2004.

[11] Peter J. Ashenden. *The Student's Guide to VHDL*. Morgan Kaufmann Publishers Inc., 1994.

[12] N Athanasiades, R Abler, J Levine, H Owen, and G Riley. Intrusion detection testing and benchmarking methodologies. In *Information Assurance Workshop,*, pages 63–72, March 2003.

[13] Michael Attig and John Lockwood. CS/CoE 535 homepage. `http://www.-arl.wustl.edu/arl/projects/fpx/cse535`, August 2003.

[14] 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)*, pages 322–323, Napa, CA, April 2004.

[15] Michael E. Attig and John W. Lockwood. Statistic Counter for Networking Hardware Modules. Technical report, WUCS-02-20, Washington University, Department of Computer Science, July 2002.

[16] Stefan Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *CCS '99: Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 1–7. ACM Press, 1999.

[17] Stefan Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Chalmers University of Technology, Sweden, Department of Computer Engineering, 2000.

[18] Zachary K. Baker and Viktor K. Prasanna. Automatic synthesis of efficient intrusion detection systems on FPGAs. In *Field Programmable Logic and Application: 14th International Conference, FPL 2004, Leuven, Belgium, August 30-September 1, 2004. Proceedings*, pages 311–321, Antwerp, Belgium, August 2004. Springer-Verlag.

[19] 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, April 2004.

[20] Zachary K. Baker and Viktor K. Prasanna. Time and area efficient pattern matching on FPGAs. In *Proceeding of the 2004 ACM/SIGDA 12th International*

Symposium on Field Programmable Gate Arrays, pages 223–232. ACM Press, 2004.

[21] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[22] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[23] Florian Braun, John W. Lockwood, and Marcel Waldvogel. Layered protocol wrappers for Internet packet processing in reconfigurable hardware. In *Proceedings of Symposium on High Performance Interconnects (HotI'01)*, pages 93–98, Stanford, CA, USA, August 2001.

[24] 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, April 2004.

[25] Young H. Cho, Shiva Navab, and William H. Mangione-Smith. Specialized hardware for deep network packet filtering. In *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 452–461. Springer-Verlag, 2002.

[26] Christopher R. Clark and David E. Schimmel. Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In *Proceedings of the Reconfigurable Computing Is Going Mainstream, 13th International Conference on Field-Programmable Logic and Applications*, pages 956–959. Springer-Verlag, 2003.

[27] Christopher R. Clark and David E. Schimmel. A hardware platform for network intrusion detection and prevention. In *Proceedings of Workshop on Network Processors and Applications (NP3)*, pages 136–145, Madrid, Spain, February 2004.

[28] 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, April 2004.

[29] Douglas Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture*. Prentice-Hall, Inc., 1988.

[30] Andy Currid. TCP offload to the rescue. *Queue*, 2(3):58–65, 2004.

[31] H. Debar. Intrusion detection products and trends. In *Proceedings of the Connect 2000*, 2000.

[32] Herve Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 32(8):805–822, April 1999.

[33] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. *SIGCOMM Comput. Commun. Rev.*, 27(4):3–14, 1997.

[34] Dorothy E. Denning. An intrusion-detection model. *IEEE Trans. Softw. Eng.*, 13(2):222–232, 1987.

[35] Neil Desai. Increasing Performance in High Speed NIDS. Online as: `http://www.snort.org/docs/Increasing_Performance_in_High_Speed_NIDS.pdf`, November 2004.

[36] Sarang Dharmapurikar, Michael Attig, and John Lockwood. Design and implementation of a string matching system for network intrusion detection using FPGA-based Bloom filters. Technical report, WUCSE-2004-12, Washington University, Department of Computer Science and Engineering, April 2004.

[37] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John W. Lockwood. Deep packet inspection using parallel Bloom filters. In *Hot Interconnects*, pages 44–51, Stanford, CA, August 2003.

[38] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. Longest prefix matching using Bloom filters. In *SIGCOMM'03*, pages 201–212, Karlsruhe, Germany, August 2003.

[39] Sarang Dharmapurikar and John Lockwood. Synthesizable design of a multimodule memory controller. Technical Report WUCS-01-26, Washington University in Saint Louis, Department of Computer Science, October 2001.

[40] M. Esmaili, R. Safavi-Naini, and J. Pieprzyk. Computer intrusion detection: A comparative survey. Technical Report 95-07, Center for Computer Security Research, University of Wollongong, May 1995.

[41] Mike Fisk and George Varghese. Fast content-based packet handling for intrusion detection. Technical report, University of California at San Diego, 2001.

[42] 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, April 2002.

[43] A.K. Ghosh, J. Wanken, and F. Charron. Detecting anomalous and unknown intrusions against programs. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'98)*, pages 259–267, December 1998.

[44] 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*, pages 404–413, Montpellier, France, 2002. Springer-Verlag.

[45] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *SIGCOMM '99: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 147–160. ACM Press, 1999.

[46] Pankaj Gupta and Nick McKeown. Packet classification using hierarchical intelligent cuttings. In *Proceedings of Symposium on High Performance Interconnects (HotI'99)*, Stanford, CA, August 1999.

[47] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[48] Juha Heinanen. Multiprotocol Encapsulation over ATM Adaptation Layer 5. Internet Engineering Task Force: RFC 1483, July 1993.

[49] Edson L. Horta, John W. Lockwood, David E. Taylor, and David Parlour. Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In *Design Automation Conference (DAC)*, New Orleans, LA, June 2002.

[50] Koral Ilgun. USTAT: A real-time intrusion detection system for UNIX. In *Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy*, May 1993.

[51] Koral Ilgun, R. A. Kemmerer, and Phillip A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Trans. Softw. Eng.*, 21(3):181–199, 1995.

[52] E. J. Johnson and A. Kunze. *IXP2400/2800 Programming: The Complete Microengine Coding Guide*. Intel Press, 2003.

[53] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[54] Christopher Kruegel, Fredrik Valeur, Giovanni Vigna, and Richard Kemmerer. Stateful intrusion detection for high-speed networks. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 285. IEEE Computer Society, 2002.

[55] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 251–261. ACM Press, 2003.

[56] James F. Kurose and Keith Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[57] Khaled Labib. Computer security and intrusion detection. *Crossroads*, 11(1):2–2, 2004.

[58] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 203–214. ACM Press, 1998.

[59] T. Lane and C. E. Brodley. An application of machine learning to anomaly detection. In *Proceedings of the 20th National Information Systems Security Conference*, pages 366–377, October 1997.

[60] M. Laubach. Classical IP and ARP over ATM. Internet Engineering Task Force: RFC 1577, January 1994.

[61] 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)*, Lecture Notes in Computer Science, Zurich, Switzerland, October 2002. Springer-Verlag.

[62] Wenke Lee and Salvatore J. Stolfo. A framework for constructing features and models for intrusion detection systems. *ACM Trans. Inf. Syst. Secur.*, 3(4):227–261, 2000.

[63] Robert Lemos. MSBlast infects eight million PCs. Online as: `http://news.-zdnet.co.uk/internet/security/0,39020375,39150721,00.htm`, April 2004.

[64] Shaomeng Li, Jim Trresen, and Oddvar Sraasen. Exploiting stateful inspection of network security in reconfigurable hardware. In *Field Programmable Logic and Applications (FPL)*, Lisbon, Portugal, September 2003.

[65] Rong-Tai Liu, Nen-Fu Huang, Chih-Hao Chen, and Chia-Nan Kao. A fast string-matching algorithm for network processor-based intrusion detection system. *Trans. on Embedded Computing Sys.*, 3(3):614–633, 2004.

[66] John W. Lockwood, James Moscola, Matthew Kulig, David Reddick, and Tim Brooks. Internet worm and virus protection in dynamically reconfigurable hardware. In *Military and Aerospace Programmable Logic Device (MAPLD)*, page E10, Washington DC, September 2003.

[67] 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)*, pages 87–93, Monterey, CA, USA, February 2001.

[68] 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)*, page 14B, Lisbon, Portugal, September 2003.

[69] John W. Lockwood, Jon S. Turner, and David E. Taylor. Field programmable port extender (FPX) for distributed routing and queuing. In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, pages 137–144, Monterey, CA, USA, February 2000.

[70] T. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P. Neumann, H. Javitz, A. Valdes, and T. Garvey (1992). A real-time intrusion detection expert system (IDES). Technical report, Computer Science Laboratory, SRI International, 1992.

[71] Bharath Madhusudan and John Lockwood. Design of a system for real-time worm detection. In *12th Annual Proceedings of IEEE Hot Interconnects*, pages 77–83, Stanford, CA, August 2004.

[72] M Mahoney and P Chan. An analysis of the 1999 DARPA/Lincoln Laboratory evaluation data for network anomaly detection. In *Proceeding of Recent Advances in Intrusion Detection (RAID)-2003*, volume 2820 of *Lecture Notes in Computer Science*, pages 220–237. Springer Verlag, September 8-10 2003.

[73] Anthony J. McAuley and Paul Francis. Fast routing table lookup using CAMs. In *INFOCOM (3)*, pages 1382–1391, 1993.

[74] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[75] David Moore, Colleen Shannon, and Jeffery Brown. Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the Internet Measurement Workshop (IMW)*, pages 273–284, 2002.

[76] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *IEEE INFOCOM 2003*, March 2003.

[77] James Moscola. Wrapper webpage. `http://www.arl.wustl.edu/arl/-projects/fpx/wrappers`, July 2001.

[78] James Moscola. FPgrep and FPsed: Packet Payload Processors for Managing the Flow of Digital Content on Local Area Networks and the Internet. Masters Thesis, Washington University in St. Louis, July 2003.

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

[80] Biswanath Mukherjee, L. Todd Heberlein, and Karl N. Levitt. Network intrusion detection. In *IEEE Network*, pages 26–41, May/June 1994.

[81] 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, April 2002.

[82] 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*, pages 1093–1097, Antwerp, Belgium, August 2004. Springer-Verlag.

[83] Rina Panigrahy and Samar Sharma. Reducing TCAM power consumption and increasing throughput. In *Proceedings of Symposium on High Performance Interconnects (HotI'02)*, pages 107–111, Stanford, CA, August 2002.

[84] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23–24):2435–2463, 1999.

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

[86] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers Inc., 2000.

[87] J. Postel. DoD Standard Internet Protocol. Internet Engineering Task Force: RFC 760, January 1980.

[88] J. Postel. DoD Transmission Control Protocol. Internet Engineering Task Force: RFC 761, January 1980.

[89] J. Postel. User Datagram Protocol. Internet Engineering Task Force: RFC 768, August 1980.

[90] N Puketza, K Zhang, M Chung, B Mukherjee, and R Olsson. A methodology for testing intrusion detection systems. *IEEE Transactions on Software Engineering*, pages 719–729, October 1996.

[91] Lili Qiu, George Varghese, and Subhash Suri. Fast firewall implementations for software-based and hardware-based routers. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 344–345. ACM Press, 2001.

[92] M.V. Ramakrishna, E. Fu, and E. Bahcekapili. A performance study of hashing functions for hardware applications. In *Proc. of Int. Conf. on Computing and Information*, pages 1621–1636, 1994.

[93] Martin Roesch. SNORT - lightweight intrusion detection for networks. In *LISA '99: USENIX 13th Systems Administration Conference on System Administration*, pages 229–238, Seattle, Washington, November 1999.

[94] 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)*, Lecture Notes in Computer Science, Berlin–Heidelberg–New York, September 2003. Springer-Verlag.

[95] David Schuehler and John Lockwood. A modular system for FPGA-based TCP flow processing in high-speed networks. In *14th International Conference on Field Programmable Logic and Applications (FPL)*, pages 301–310, Antwerp, Belgium, August 2004.

[96] 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.

[97] David V. Schuehler and John Lockwood. TCP-Splitter: A TCP/IP flow monitor in reconfigurable hardware. In *Hot Interconnects*, pages 127–131, Stanford, CA, August 2002.

[98] 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, pages 301–310, Antwerp, Belgium, August 2004. Springer-Verlag.

[99] David V. Schuehler, James Moscola, and John W. Lockwood. Architecture for a hardware based, TCP/IP content scanning system. In *Hot Interconnects*, pages 89–94, Stanford, CA, August 2003.

[100] Steven J. Scott. Snort Installation Manual. Online as: `http://www.snort.-org/docs/snort-rh7-mysql-ACID-1-5.pdf`, August 2002.

[101] Michael Sebring, Eric Shellhouse, Mary Hanna, and R. Alan Whitehurst. Expert systems in intrusion detection: A case study. In *Proceedings of the 11th National Computer Security Conference*, pages 74–81. National Institute of Standards and Technology/National Computer Security Center, October 1988.

[102] Stanislav Shalunov and Benjamin Teitelbaum. TCP use and performance on Internet2. In *Proceedings of ACM SIGCOMM Measurement Workshop*, 2001.

[103] R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2001.

[104] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 213–224. ACM Press, 2003.

[105] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur. DIDS (Distributed Intrusion Detection System) - Motivation, architecture and an early prototype. In *Proceedings of the 14th National Computer Security Conference*, pages 167–176, October 1991.

[106] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the 10th ACM Conference on Computer and Communication Security*, pages 262–271. ACM Press, 2003.

[107] Haoyu Song. Secure remote control and configuration of FPX platform in gigabit ethernet environment. Technical report, WUCSE-2003-68, Washington University in St. Louis, 2003.

[108] 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)*, pages 238–245, Monterey, CA, February 2005.

[109] Haoyu Song, Jing Lu, John Lockwood, and James Moscola. Secure remote control of field-programmable network devices. In *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, pages 334–335, Napa, CA, April 2004.

[110] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, large-scale string match for a 10 Gbps FPGA-based network intrusion detection system. In *Proceedings of the Reconfigurable Computing Is Going Mainstream, 13th International Conference on Field-Programmable Logic and Applications*, pages 880–889. Springer-Verlag, 2003.

[111] 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, April 2004.

[112] Todd Sproull, John W. Lockwood, and David E. Taylor. Control and configuration software for a reconfigurable networking hardware platform. In *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, April 2002.

[113] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. *SIGCOMM Comput. Commun. Rev.*, 28(4):191–202, 1998.

[114] V. Srinivasan and George Varghese. Faster IP lookups using controlled prefix expansion. In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 1–10. ACM Press, 1998.

[115] 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*, pages 484–493, Antwerp, Belgium, August 2004. Springer-Verlag.

[116] Kymie M. C. Tan and Roy A. Maxion. "Why 6?" Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 188–202. IEEE Computer Society, May 2002.

[117] Andrew S. Tanenbaum. *Computer Networks (3rd ed.)*. Prentice-Hall, Inc., 1996.

[118] David Taylor. Survey & taxonomy of packet classification techniques. Technical Report WUCSE-2004-24, Washington University in Saint Louis, July 2001.

[119] David E. Taylor, John W. Lockwood, and Naji Naufel. RAD module infrastructure of the field-programmable port extender (FPX). Technical Report WUCS-TM-01-16, Applied Research Laboratory, Department of Computer Science, Washington University in Saint Louis, July 2001.

[120] David E. Taylor, Jonathan S. Turner, John W. Lockwood, Todd S. Sproull, and David B. Parlour. Scalable IP lookup for Internet routers. *IEEE Journal on Selected Areas in Communications (JSAC)*, 21(4):522–534, May 2003.

[121] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection. In *Proceedings of the IEEE Infocom Conference*, Hong Kong, China, March 2004.

[122] J. Turner, T. Chaney, A. Fingerhut, and M. Flucke. Design of a Gigabit ATM Switch. In *Proceedings of Infocom 97*, March 1997.

[123] Beverly Waite. Malicious Code Attacks Had $13.2 Billion Economic Impact in 2001. Online as: `http://www.computereconomics.com/article.cfm?id=133`, January 2002.

[124] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed IP routing table lookups. In *Proceedings of ACM SIGCOMM '97*, pages 25–36, September 1997.

[125] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high-speed prefix matching. *ACM Trans. Comput. Syst.*, 19(4):440–482, 2001.

[126] Sudhakar Yalamanchili. *Introductory VHDL From Simulation to Synthesis.* Prentice-Hall, Inc., 2001.

[127] Tak W. Yan and Hector Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Trans. Database Syst.*, 19(2):332–364, 1994.

[128] F. Yu and R. Katz. Efficient multi-match packet classification and lookup with TCAM. In *12th Annual Proceedings of IEEE Hot Interconnects*, Stanford, CA, August 2004.

[129] Fang Yu, Randy H. Katz, and T. V. Lakshman. Gigabit Rate Packet Pattern-Matching Using TCAM. In *ICNP '04: Proceedings of the Network Protocols, 12th IEEE International Conference on (ICNP'04)*. IEEE Computer Society, 2004.

[130] Cliff Changchun Zou, Lixin Gao, Weibo Gong, and Don Towsley. Monitoring and early warning for internet worms. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 190–199. ACM Press, 2003.

Vita

Michael E. Attig

Date of Birth June 25, 1981

Place of Birth Portland, Oregon

Degrees
B.S. Summa Cum Laude, Computer Engineering, May 2003
B.S. Summa Cum Laude, Electrical Engineering, May 2003

Honor Societies
Tau Beta Pi (Engineering Honor Society)
Eta Kappa Nu (Electrical Engineering Honor Society)
IEEE (Institute of Electrical and Electronics Engineers)

Publications

A Framework for Rule Processing in Reconfigurable Network Systems, by Michael Attig and John Lockwood; *In Proceedings of: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, April 18-20, 2005.

Transformation Algorithms for Data Streams, by John W. Lockwood, Stephen G. Eick, Doyle J. Weishar, Ron Loui, James Moscola, Chip Kastner, Andrew Levine, and Michael Attig; *In Proceedings of: IEEE Aerospace Conference*, Big Sky, Montana, March 2005.

Implementation Results of Bloom Filters for String Matching, by Michael Attig, Sarang Dharmapurikar, and John Lockwood; *In Proceedings of: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, April 20-23, 2004.

Technical Reports

Design and Implementation of a String Matching System for Network Intrusion Detection using FPGA-based Bloom Filters, by Sarang Dharmapurikar, Michael Attig, and John Lockwood; *WUCSE-2004-12* (April 2004).

Usage of the Statistics Counter Plus Component in Networking Hardware Modules, by Michael Attig and John Lockwood; *WUCSE-2002-25* (August 2002).

Implementation of a Pipelined Control Cell Processor, by Michael Attig and John Lockwood; *WUCSE-2002-24* (August 2002).

Statistics Counter for Networking Hardware Modules, by Michael Attig and John Lockwood; *WUCSE-2002-20* (July 2002).

May 2005