ABSTRACT

ALDWAIRI, MONTHER MUSTAFA. Hardware Efficient Pattern Matching Algorithms and Architectures for Fast Intrusion Detection. (Under the direction of Dr. Paul Franzon).

Intrusion detection processors are becoming a predominant feature in the field of network hardware. As demand on more network speed increases and new network protocols emerge, network intrusion detection systems are increasing in importance and are being integrated in network processors. Currently, most intrusion detection systems are software running on a general purpose processor. Unfortunately, it is becoming increasingly difficult for software based intrusion detection systems to keep up with increasing network speeds (OC192 and 10Gbps at backbone networks).

Signature-based intrusion detection systems monitor network traffic for security threats by scanning packet payloads for attack signatures. Intrusion detection systems have to run at wire speed and need to be configurable to protect against emerging attacks. This dissertation describes the concept, structure and algorithms for a special purpose hardware accelerator designed to meet those demands. We consider the problem of string matching which is the most computationally intensive task in intrusion detection. A configurable string matching accelerator is developed with the focus on increasing throughput while maintaining the configurability provided by the software intrusion detection systems. A hardware algorithm for efficient data storage and fast retrieval is used to compress, store and retrieve attack signatures. Our algorithms reduce the size of the rules to fit on chip and enables intrusion detection to run at line rates and faster.

HARDWARE-EFFICIENT PATTERN MATCHING ALGORITHMS AND ARCHITECTURES FOR FAST INTRUSION DETECTION

by

MONTHER ALDWAIRI

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

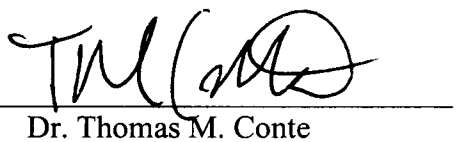
COMPUTER ENGINEERING

Raleigh

2006

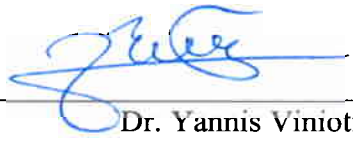
APPROVED BY:

Dr. Paul Franzon
Chair of Advisory Committee



Dr. Thomas M. Conte

Dr. George Rouskas



Dr. Yannis Viniotis

Dedication

to myself…

Biography

MONTHER ALDWAIRI was born on February 26, 1976, in Irbid, Jordan. He grew up in Kittim, graduated from Khalil Al-Salem High School in Al-Husin. Monther received his Bachelors of Science in Electrical Engineering from Jordan University of Science and Technology in 1998. After graduation, Monther worked for ARAMEX International as Systems Integration Engineer. In 2000, he followed on the footsteps of his older brothers and sisters and moved to the United States to finish his graduate studies. He joined the department of Electrical and Computer Engineering at North Carolina State University. He received the Master's degree in Computer Engineering in 2001 and the Doctor of Philosophy degree in 2006. Monther's research interests include but not limited to Network Processors, Network Security, Intrusion Detection, Pattern Matching, Hardware Algorithms, and Digital Design.

Upon completing his Ph.D., Monther will be joining Jordan University of Science and Technology as an assistant professor.

Acknowledgments

First, I am deeply indebted to my supervisor Professor Paul Franzon whose guidance, inspiration, and suggestions helped me finish this work. Throughout my years in school, he provided encouragement, sound advice, good teaching, and lots of good ideas. I am also grateful to my committee members: Dr. Thomas Conte, Dr. Yannis Viniotis, Dr. George Rouskas, Dr. Beverly Taylor and Dr. Gregory Byrd.

I would like to thank all of the members of the Electronic Research Laboratory (ERL) for always keeping things interesting, particularly those whom I have worked with in the past and those who have contributed to the Special Processors and Microsystems Group. I would like to thank Mohamed Guled, Ambrish Varma, Lei Luo, Meeta Yadav and Steven Mick.

I am grateful to all my friends who made the times I spent in North Carolina both rewarding and enjoyable. For all the time we spent at Cup-a-Joe discussing anything and every thing, I would like to thank Khaled Gharaibeh and my brother

Hazim Aldwairi. I have to furthermore thank my great friends Muawyeh Al-Otoom, Tariq Gaith, Mazen Kharbutli, Ali El-Haj-Mahmoud, Ahmed Alzawawi, Fatima Al-Harbi and Saba Kawas. Many thanks to: Chen Qi, Ali Shayanfer, Bin Li and Phoemphun Oothongsap.

Special thanks go to the members of NC State, NCTTA, Cary TTA and Cary TTC table tennis clubs and organizations. To all my soccer buddies at the Miller fields, Derr tracks, Lake Boone High and NCSU intramural soccer, thank you.

Finally and most importantly, I am forever indebted to my parents and my family for their unconditional love, endless patience and encouragement when it was most required.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Intrusion detection is becoming a necessary task in network processing. New activities that involve the tracking of U.S. and allied network traffic and online shopping increased the demand for secure networks. Wireless computing, cell phones being equipped with network connectivity and multimedia application have increased the demand on network bandwidth. As demand on more network speed increases and new network protocols emerge, Network Intrusion Detection Systems (NIDS) are increasing in importance and are being integrated in network processors.

Many of today's networks require, or would benefit from, the inclusion of intrusion detection. Currently, most Intrusion Detection Systems (IDSs) are software running on a general purpose processor. Unfortunately, it is becoming increasingly difficult for software based IDSs to keep up with increasing network speeds (OC192 or 10Gbps at backbone networks). Application Specific Integrated Circuits (ASICs) used in networks today are inflexible and lack the security needed to combat emerging threats. This has prompted the

need to accelerate intrusion detection while maintaining the configurability needed to detect new attacks.

1.1 Scope

This dissertation outlines the development of an intrusion detection system that is capable of efficiently reconfiguring itself, while matching the speeds required by today's networks. This effort is driven by the need for high performance and flexibility in network intrusion detection. The focus is on signature-based intrusion detection systems and specifically on the most computationally expensive side of the intrusion detection problem, that is pattern matching. The solution is presented here in the form of a high speed network intrusion detection processor, which relies on special pattern matching algorithms and unique architectures to reduce the memory requirements and achieve the required throughput. The solution includes a complier that will read the attack signatures and generate compressed state tables to initialize the architecture. The algorithms employ novel techniques to remove all redundancies in the state tables to reduce its size to fit on-chip. To demonstrate those features, a complete theoretical and experimental analysis is presented.

1.2 Problem Overview

The explosion of recent attacks by CodeRed and MSBlast affected the productivity of computer networks all over the world. It is difficult to precisely estimate the damages caused by viruses and worms. The damages may include the loss of information, productivity and customer confidence. With the Internet traffic doubling every six months, hacks, attacks and viruses have increased as well, increasing the need for network security [1]. Everyone is starting to recognize the increasing threat caused by cyber attacks and is taking measures to effectively detect and respond to those attacks.

Traditionally, networks have been protected using firewalls that provide the basic functionality of monitoring and filtering traffic. Firewall users can then write rules that specify the combinations of packet headers that are allowed through. However, not all incoming malicious traffic can be blocked and legitimate users can still abuse their rights. Intrusion detection systems go one step further by inspecting packet payload for attack signatures. Currently, the majority of IDSs are software applications running on a general purpose processor and standard Microsoft Windows or Linux operating systems. These platforms provide sufficient power to capture and process data packets at speeds of only 100 Mbps. Consequently, most intrusion detection systems today are running offline, analyzing the traffic after it's allowed into the network. Alerts are sent to the security engineer identify attacks after they already happened.

Offline detection of suspicious activity is not enough and systems that protect networks are needed. Intrusion Prevention Systems are able to protect the network from ongoing attacks. They detect malicious packets and stop intrusions by dropping packets or resetting connections. Intrusion prevention systems automatically block the suspicious traffic before it does any damage rather than simply raising an alert. Therefore, packets need to be processed in real-time or near real-time. Unfortunately, none of the systems available today can thwart attacks effectively at line rates. This has prompted the need for faster intrusion detection and prevention systems.

In an effort to meet the needs of today's network security, academic researchers have been active in creating new hardware architectures to speed up intrusion detection in general and pattern matching specifically. Several hardware accelerators have been proposed. Early efforts focused on using Deterministic Finite Automata (DFA) and mapped it directly on a Field Programmable Gate Array (FPGA) [2,3]. The need to rebuild the FA and reprogram the FPGA every time a rule is added or deleted made the FA implementations impractical. They also achieved modest throughput due to the large size of the FA and consequently failed to achieve line rates. More recent efforts included the use of Content Addressable Memories (CAM) [4,5] and discrete comparators [6,7] (see Section 2.4 for a complete survey of the related work). Both the CAM and discrete comparators implementations achieved higher throughput, but not enough to run intrusion detection at backbone network speeds. Both designs suffered from large area and power requirements. In general, these IDS accelerators

are intended to provide more flexibility than custom ASICs coupled with more throughput than software IDSs.

Unfortunately, most of the proposed failed to provide enough throughput to support the increasing network speeds or real time packet inspection. The systems that offered higher speeds sacrificed configurability as well as area and power requirements.

1.3 Solution Synopsis

The move to hardware based systems allows the introduction of more parallelism than possible in software based systems. Hardware based solutions are probably the only approach currently practical to provide both the flexibility and performance required by current cutting-edge and future networks. A hardware based intrusion detection accelerator can provide intrusion detection and string matching at Giga-bit speeds, allowing inline packet inspection. It can be easily reconfigured to protect against emerging attacks and can adapt its structure to support new security features required by a particular network. Furthermore, the device configurable structure and pattern matching algorithms can be adapted to perform other networking tasks.

A configurable memory-based string matching accelerator is developed with the focus on increasing throughput while maintaining the configurability provided by the software

intrusion detection systems. Although ASICs are the best option for matching the speed requirement, they are inflexible and very expensive to replace. With FPGA processing power on the rise and some chips housing embedded processor cores; FPGAs could be a better option today. FPGAs are easier to build, reprogram and make the task of creating a proof of concept prototype easier. Here we propose that programmable logic devices and memories be used to accelerate intrusion detection, as opposed to all software or ASICs. FPGA-based platforms can exploit the fact that the NIDS rules change relatively infrequently, and use reconfiguration to reduce implementation cost. In addition, FPGA-based systems can exploit parallelism in order to achieve satisfactory processing throughput.

Special pattern matching algorithms are developed to reduce the memory required to store the attack signatures to fit on-chip. A Special architecture is built to utilize those algorithms and allow online reconfiguration. The size of the attack signatures is significantly reduced so that an external high speed SRAM might be used to boost performance. The memory based accelerator is composed of a software based component that runs on a general purpose processor, and a standard RAM based technique for FSM implementation. The software generates an FSM from the set of strings extracted from the Snort rule database. The FSM matches multiple strings at the same time based on the Aho-Corasick string matching algorithm (see Section 2.3). The software converts the FSM into state tables that are mapped onto the RAM. Matching an incoming packet against the attack signatures is as simple as a memory access. The RAM contributes to the accelerator flexibility, configurability and ease of use. The throughput increases as the on-chip RAM bandwidth increases. The accelerator

utilizes special hardware efficient algorithms for data compression and fast retrieval. They enable the system to reduce the amount of on-chip memory needed and to speed up the memory access time. The algorithms simplify and accelerate the matching process and allow higher throughput.

The investigation and development of such an intrusion detection processor is presented here, with a focus on the added benefits of throughput, configurability, and reduced memory requirements. A full description of the architecture and algorithms is presented. In addition, a detailed performance analysis and experimental results are used to demonstrate the higher throughput and smaller memory requirements.

1.4 Contributions and Claims

The objective of this effort is to demonstrate the viability of a high-speed reconfigurable string matching accelerator for intrusion detection. The primary challenge faced is the development of a hardware architecture that supports reconfiguration while maintaining the performance needed by the state of the art networks of today. The architecture is flexible and uses a memory to store the attack signatures in the form of state tables. This simplified the pattern matching process to a series of memory reads. It also enabled easy and fast reconfiguration in the form of memory writes compared to the expensive FPGA reprogramming. The system utilized a simple classification algorithm that exploited the

mutual independence between the different packet streams to run multiple units at the same time. That enabled higher throughput and allowed reconfiguring one unit while keeping the other units online.

To support this effort, hardware algorithms were created to extract signatures and produce compressed tables to program the accelerator memory, and a novel approach for providing reconfiguration information to the system was devised. This includes the development of special pattern matching algorithms suitable for hardware implementations. In addition, the effort included the development of hardware compression algorithm that reduces the large memory required to store the attack signatures. The primary benefit attained through this, which advance the state-of-the-art, is a fast reconfigurable network intrusion detection system that allows for line rate network protection and compresses the attack signatures that fit on-chip. The concept of pattern matching presented here is not limited to intrusion detection, however. It may be directly applied to any computing task requiring pattern matching such as IP forwarding and packet classification.

The strengths and weaknesses of the algorithms and architecture are analyzed, and experimental results are examined to determine the effectiveness of the implementation. The resulting contributions of this work are threefold:

1. Pattern matching coprocessor to speed up intrusion detection with a reconfigurable design that can adapt to changing attacks and changing protocols.

2. Hardware specific pattern matching algorithms for fast state table storage, compression and retrieval.
3. Software (complier) that parses the strings, partitions FSMs, creates state tables and compresses the data.

Specifically, this work makes the following research contributions:

- We describe a novel configurable string matching architecture that can operate at speeds of more than 10 Gbps.
- We present a novel concept for splitting large FSMs to several small FSMs that can run in parallel. They key to reduce the size of the database and achieve higher speeds is to convert the FSM to a tabular format and program it into a RAM which can easily be accessed through special hardware. In addition to that, we split the large FSM into several small FSMs that can be accessed in parallel to achieve higher matching speeds.
- We present a novel string matching algorithm that can store the Snort rule set in only 0.36 MB. The algorithm is based on compressing all the redundant information in the state tables such as failure pointers to the idle states. The size of the Snort rules stored in the software implementations is close to 58 MB and the best compression technique described in the literature required more than 1 MB of RAM.

- The compiler is a novel piece of software that parses the strings, partitions FSMs, creates state tables and compresses the data. The compiler takes only in the order of tens of milliseconds to complete.

1.5 Dissertation Outline

This document describes a solution to the problem presented in Section 1.2. Chapter 2 provides a background on string matching algorithms, summarizes IDS acceleration efforts and points out some of the differences between the presented accelerator and the architectures proposed in the literature. In Chapter 3, an overview of the structure and operation of the string matching accelerator including the hardware architecture, software, complier and interface is presented. Chapter 4 discusses in details the concept of splitting FSMs, classification algorithms, load balancing, the modified pattern matching algorithms and analyzes the proposed hardware specific compression algorithms for pattern matching. Chapter 5 presents the simulation results, discusses the Snort rules, measures the memory requirement, analyzes the performance of the accelerator, measures the configuration time, presents the updated performance of the compression algorithm and compares the results with the previous work. Finally, Chapter 6 summarizes the findings and provides some insight into future directions.

Chapter 2

STATE OF THE ART REVIEW

This chapter provides an introduction to network security and intrusion detection in Sections 2.1 and 2.2, respectively. Section 2.3 introduces two of the most widely used pattern matching algorithms. Section 2.4 contains an overview of the current state of the art in the area of intrusion detection. Section 2.5 summarizes the background, and lists the shortfalls of the current intrusion detection technologies.

2.1 Network Security

Network security is a dynamic field; new attacks are catching the headlines every year. Internet attacks have great impact on network and business productivity whether they are successful or not. If a network is compromised, there is a loss of productivity, sales or confidential information. The loss of money could also be in the form of allocating increased

budgets to counter serious threats. The question is how to effectively protect against network attacks? A traditional firewall alone is not enough to protect the network, nor is encryption. To secure a network, a combination of IDSs, encryption and access control must be used. Having a knowledgeable security engineer is a must, therefore, in this section we look at the most common attacks and how to defend against them.

2.1.1 Denial of Service Attacks (DoS)

A "denial of service" attack is an attempt by attackers to prevent legitimate users of a service from using that service, such as email or a popular website. It can be executed by consuming a limited resource, destroying/changing configuration information or physical destruction of network components. A very common example is the SYN flood attack, which is simply to send a large number of SYN packets and never acknowledge any of the replies. This leads the recipient to storing more records of SYN packets than it can handle. A technical fix was introduced, called SYNcookie, where recipients don't keep information of half opened connections anymore. Instead they send back an encrypted version of their sequence number and wait for the host to acknowledge before the connection is stored.

A new type of DoS attacks has surfaced lately; it's called Distributed Denial of Service Attack (DDoS). An attacker subverts a large number of machines over a period of time, and installs custom attack software in them. At a predetermined time, or on a given signal, these

machines (called zombies) start to bombard the target site with messages [8]. Those attacks may sound complicated and that they require a deep knowledge of hacking techniques to execute them. Unfortunately, that is not the case; tools such as Trinoo [9] are ready and available online for script kiddies to download and use.

2.1.2 IP Spoofing

A remote machine acts as a node on the local network, finds vulnerabilities with the servers and installs a backdoor program to gain control over network resources. An attacker takes over a local host (X) down with some DoS attack, and then initiates a new connection with another host (Y) pretending to be X. He can do that by guessing Y's sequence number, although modern stacks use random number generators and other techniques to avoid this predictability. But several tools are available to assist crackers in performing such vulnerability.

IP Spoofing is not easy to defend against, since it's tightly related to the design of the TCP/IP suite. At the firewall level, the incoming packets with private IP addresses on the local network should be blocked. Additionally, the firewall should not accept addresses with the network internal range as the source IP. For the outgoing packets, do not allow source addresses outside of your valid range, which will prevent some compromised machine on

your network from sending spoofed traffic to the Internet. Hastings et al. [10] provides more details on how to defend against IP spoofing.

2.1.3 Viruses, Trojans and Worms

A virus is a program or piece of code that is loaded onto a computer and runs without the owners' knowledge. Some are simply annoying, while others can be very destructive. Most viruses replicate themselves by attaching themselves to other programs. A worm tries to gradually infect as many computer files as possible, they also duplicate by attaching themselves to email, or other file transfers. Worms may seem less destructive compared to viruses, but they harm the network by spreading at higher rates affecting the productivity. Worms might also contain payloads that could delete files on a host system, send confidential files through email and install backdoors or Trojans. A Trojan is a destructive program that pretends to be a useful one. The Trojan contains the virus code, which then infects the computer. So, although the Trojan itself might not be self-replicating, the virus it contains is.

Viruses, worms and Trojans mainly spread by exploiting vulnerabilities in operating systems and installed software. They also might spread by tricking users to assist them. Protecting against viruses involves educating the users to only download files from trusted sources. Users need not to be opening suspicious email attachments, and certainly should not run attached files or programs. All vendors supply regular security updates to their software

and operating systems. These need to be installed regularly on all network machines to prevent virus infections. Anti-virus and anti-spyware software are an important components in defending against viruses. There are many virus scanners and protection programs available, both in open source and commercial versions. These programs contain signatures of viruses, for which they scan the computer files, and that must be kept up to date. For regular updates on viruses, worms and Trojans refer to Symantec [11] and McAfee [12] online virus libraries. Intrusion detection systems can be extended to perform virus detection.

2.1.4 Stack Overflow Attacks

A buffer is a contiguous allocated block of memory, such as an array or a pointer. In C and C++, there are no automatic bounds checking on a buffer, which means a user can write past a buffer end point. A stack is a buffer that contains subroutine parameters and return addresses. A stack pointer (SP) points to the top of the stack. Whenever a function call is made, the function parameters are pushed onto the stack from right to left. Then the return-address (address to be executed after the function returns), followed by a frame pointer (FP), is pushed on the stack. A frame pointer is used to reference the local variables and the function parameters, because they are at a constant distance from the FP. Local automatic variables are pushed after the FP. In most implementations, stacks grow from higher memory addresses to the lower ones. All these variables are cleaned up from the stack as the function terminates. A program that writes beyond an allocated buffer can cause unexpected behavior

by over writing the space allocated to the FP, return address, etc. The C function strcpy() which doesn't check bounds was used to overwrite a function's return address, which in turn can alter the program's execution path. An intelligent hacker might want to spawn a shell (with root permissions) by jumping the execution path to such code.

The defense against buffer overflows starts with code scrutiny or writing secure code. Programmers should avoid the use of unsafe functions such as strcpy(), strcat() or any other functions that don't check for boundaries. Compilers play a big rule in generating warnings against the use of unsafe functions. No complier warning should be taken lightly. Finally, runtime boundary checking is the last line of defense against buffer overflow attacks. Runtime boundary checking [13] adds instructions to check array bounds and performs pointer checking in run time. Several approaches have been developed to check code for buffer overflow vulnerabilities [14].

2.1.5 Summary

Observe that none of these attacks discussed above is stopped by encryption, and not all of them by firewalls. Most of the exploits make use of application or operating system bugs, of which the majority, are stack overflow vulnerabilities. The obvious solution is to fix the system bugs, but as systems become more complicated, it is hard to keep up with all the vulnerabilities that are discovered in operating systems and network protocols. In fact, there

is a race between the attackers, who try to find loopholes, and the vendors, who develop patches for them. Attackers are creating attack tools and posting them online for unskilled hackers to use. In effect, hacking is being deskilled, while defense is becoming more complex. Having multiple lines of defense is the best strategy to protect networks; a combination of IDSs, encryption and access control must be used. United States Computer Emergency Readiness Team (US-CERT) [15] is an excellent resource on all information related to network security, including security alerts, advisories and virus resources. The next subsection focuses on intrusion detection which is a very important line of defense against security threats.

2.2 Intrusion Detection

Intrusion detection is the process of monitoring and analyzing activities in a computer system or network to find intrusions. Intrusions are activities that violate the security policy of the computer system or network. The intruders may be local users misusing their privileges or Internet hackers trying to gain access to restricted resources. Intrusion detection is particularly important because computer systems are designed without security in mind or they might have bugs or weaknesses that could be exploited by an intruder.

Intrusion detection systems can run in one of several modes: intrusion detection, inline IDS or honeypot mode. In intrusion detection mode, the IDSs monitor the traffic offline and

draw the attention of network administrator to suspicious activity by sending alerts [16]. An inline intrusion detection system or Intrusion Prevention System (IPS) actively filters exploits from traffic in real-time. It can forge resets, drop packets, or modify the packets in transit to defeat an attack. IPSs have to be extremely fast and reliable to process packets in real-time and should be completely transparent, so there is no need to change the network configuration. In honeypot mode, IDSs act as a closely monitored network decoy serving several purposes, they can distract attackers from more valuable machines on a network. In addition, they can provide early warning about new attacks and they allow the examination of attacks during and after exploitation of the honeypot. Honeypots are becoming increasingly popular and open-source versions are available for Windows and Linux [17,18].

Intrusion detection systems have been traditionally classified into host based, and network based systems. Host based IDSs detect attacks against a single host. Network based IDSs analyze network traffic to detect attacks on all computers connected to the network. The NIDSs can be further segmented into one of two techniques: anomaly detection or misuse detection (signature based). Anomaly detection is based on searching for discrepancies from the models of normal behavior. These models are obtained by performing a statistical analysis on the history of system calls [19,20,21] or by using rule based approaches to specify behavior patterns [22,23]. Signature based or misuse detection is based on searching packets for attack signatures. Rules are manually added to specify signatures for new attacks. Misuse detection is much faster than anomaly detection, but can detect only those attacks that already have signatures. On the other hand, anomaly detection

have the advantage of being able to detect previously unknown attacks, however it suffers from a large number of false positives.

Misuse detection can be also stateful by considering attacks spread over several packets. Stateful IDS examines the packet content up to the application layer, monitors the state of a connection and maintains historic information in a state table. This approach is more capable of detecting advanced attacks but is much more expensive in terms of computing and memory requirements making it a favorite target for denial of service attacks.

2.3 String Matching Algorithms

At the core of every intrusion detection system is a string matching algorithm. String matching is the main task in intrusion detection. From a stream of packets, the algorithm identifies those packets that contain data matching the signatures of a known attack. The intrusion detection system then takes action that could vary from alerting the system administrator to dropping the packet in the case of inline IDS. The problem of pattern matching is well researched, many algorithms exist and they can be classified into either single pattern string matching or multiple pattern string matching. In single pattern string matching the packet is searched for a single string at a time. On the other hand, in multiple pattern string matching the algorithm searches the packet for the set of strings all at once.

Most known intrusion detection systems use a general purpose string matching algorithm such as Boyer-Moore (BM) [24]. BM is the most widely used algorithm for string matching in intrusion detection, the algorithm compares the string to the input starting from the rightmost character of the string. To reduce the large number of comparisons, two heuristics are triggered on a mismatch. The bad character heuristic shifts the search string to align the mismatching character with the rightmost position at which the mismatching character appears in the search string. If the mismatch occurs in the middle of the search string, then there is suffix that matches. The good suffix heuristic shifts the search string to the next occurrence of the suffix in the string. Fisk and Varghese suggested a set-wise Boyer-Moore-Horspool algorithm specifically for intrusion detection [25]. It extends BM to match multiple strings at the same time by applying the single pattern algorithm to the input for each search pattern. Obviously this algorithm does not scale well to larger string sets.

On the other hand, Aho-Corasick (AC) [26] is a multiple pattern string matching algorithm, meaning it matches the input against multiple strings at the same time. Multiple pattern string matching algorithms generally preprocess the set of strings, and then search all of them together over the packet content. AC is more suitable for hardware implementation because it has a deterministic execution time per packet. Tuck et al. [27] examined the worst-case performance of string matching algorithms suitable for hardware implementation. They showed that AC has higher throughput than the other multiple string matching algorithms and is able to match strings in worst-case time linear in the size of the input. They concluded

that their compressed version of AC is the best choice for hardware implementation of string matching for IDS.

Aho-Corasick works by building a tree based state machine from the set of strings to be matched as follows. Starting with a default no match state as the root node, each character to be matched adds a node to the machine. Failure links that point to the longest partial match state are added. To find matches, the input is processed one byte at a time and the state machine is traversed until a matching state is reached. Figure 2.1 shows a state machine constructed from the following strings {hers, she, the, there}. The dashed lines show the failure links, however the failure links from all states to the idle state are not shown. This gives an idea of the complexity of the FSM for a simple set of strings.

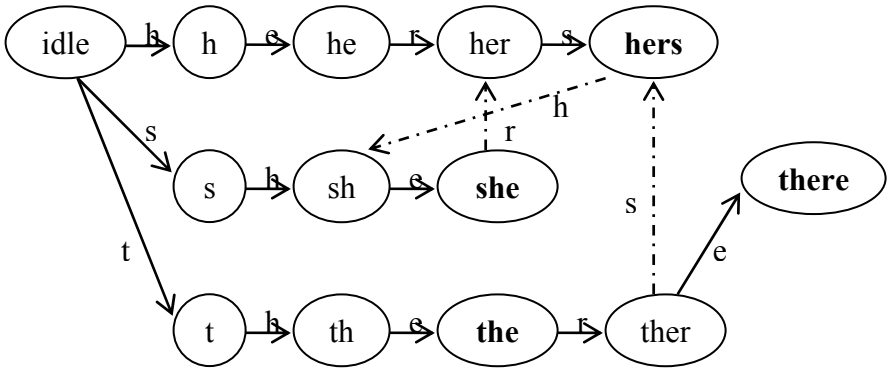


Figure 2.1. Finite state machine diagram

2.4 Current State of the Art for Intrusion Detection

Network intrusion detection is an active research field both academically and commercially. Commercial systems are mostly software running on personal computers stripped out of all unnecessary features that may be vulnerable to attacks. Commercial systems are continuously offering new features such as stateful inspection and intrusion prevention. At the same time, academic research continues into new methods of providing IDSs in reconfigurable hardware. This section identifies the most widely used corporate IDSs and discusses current academic research directed toward moving intrusion detection into reconfigurable hardware.

2.4.1 Commercial Intrusion Detection Systems

There is a large number of commercial and open-source intrusion detection systems. It would be impractical to survey them all. On the other hand, open-source intrusion detection is becoming increasingly popular with Snort [28] becoming the de facto standard for intrusion detection system. Another popular open-source intrusion detection system is Shadow [29] which released under GNU software license. Most of the configurable hardware developed by academic research used Snort as the standard intrusion detection system. We selected Snort because it is open source and because there is a vast amount of information available. It would also make it easier to compare to other architectures. The next subsection presents a detailed discussion of Snort.

2.4.1.1 Snort

Snort [28] is a widely used open-source intrusion detection system in which the rules refer to the header as well as to the packet payload. Snort also allows stateful analysis to be performed as explained in [30]. Snort signatures databases are freely available and maintained online [31]. Snort rules are divided into two sections, the rule header and the rule options. The header contains the rule's action, protocol, source and destination IP addresses including network masks, and the source and destination ports. The options section is optional, it follows the rule header and is enclosed inside a pair of parentheses. There may be one option or many, and the options are separated with a semicolon. If a rule has multiple options, these options form a logical AND. The action in the rule header is invoked only when all criteria's in the options are true. All options are defined by keywords specifying which fields of the packet should be inspected, such as *ttl* and *content*. The *content* keyword contains a signature or data pattern to be matched against the packet content. The pattern may be in the form of an ASCII string or a binary data in the form of hexadecimal characters. The ASCII string is generally enclosed in a pair of double quotes ("") and the hexadecimal characters are enclosed in a pair of pipe or vertical bar symbols (||). If the signature includes a double quote, semicolon or pipe characters, they must be escaped [32].

There are other keywords that are used with the *content* keyword to add additional criteria while finding a pattern inside a packet, such as the *offset, depth*, and *nocase* keywords. Using the *offset* keyword, you can start your search at a certain offset from the

start of the packet payload. The *depth* keyword is used to define the point after which Snort should stop searching for the pattern in the packets payload. The search is always case sensitive unless the *nocase* keyword is used to indicate the opposite. The *dsize* keyword is used to test if the packet payload exceeds a certain set limit in case of buffer overflow attacks. The *msg* keyword is used to add a text string for logs or alerts. The *rev* keyword indicates the revision number for the rule. *The uricontent* keyword is used to look for a string only in the URI part of a packet. For more details about Snort rules options check chapter three of R. Rehman's book [33].

```
alert tcp $TELNET_SERVERS 23 -> $EXTERNAL_NET any
(msg:"TELNET root login"; content:"login\: root";
low:from_server, established; classtype:suspicious-
login;sid:719; rev:5)
```

Figure 2.2. Sample Snort rule

A sample Snort rule that detects Telnet root login attempt is shown in Figure 2.2. The action to be taken is to send an alert message, other actions could be to log the packet or just ignore (pass) it. Any packet from local telnet server port 23 destined to any external host would match the rule's header. If the rule's header matches, the rule's options are then

examined. The *msg* keyword contains the message to be logged "TELNET root login". The *content* keyword indicates the signature to be matched against the packet content. The *sid* keyword indicates the vulnerability identification number assigned to this rules. The number is used to get additional information, from the arachNIDS web database [31] for example.

Snort 2.0 is written in C, and is composed of three main components: the packets decoder, detection engine and alerter or logger. The packet decoder performs all the parsing to prepare the packet data for the detection engine. The detection engine is at the heart of Snort. The engine is broken into three processing phases: The rule optimizer, the multi-rule matching engine and the event selector. The rule optimizer optimizes the Snort rules by sorting them into smaller sets. The packets are classified based on the header fields and assigned to a specific rule subset. This allows Snort to inspect the packet against the applicable rule set. The rules subsets are parsed and stored in a two dimensional linked list:

1. Chain Headers
2. Chain Options

These rule chains are searched recursively for each packet. If the matching engine triggers a match an event is generated and added to the event queue. The event selector prioritizes the events and processes the queue. The Logger or alerter then executes the action specified in the rule definition. The logger allows the information to be logged in *tcpdump* format. The alerts can be sent to a *syslog*, file, UNIX sockets or a database. By default, all logs are written in the *"/var/log/Snort*" folder and all of the alerts are written to the "*/var/log/Snort/alerts*" file [34].

Snort can be run as a packet sniffer, packet logger and as a network intrusion detection system. When Snort is run as a packet sniffer, the packet header information and data are dumped on the standard output. As a packet logger, Snort logs the application and protocol header information to a file. As an NIDS, Snort listens on the network interface and runs Snort detection engine.

Based on the previous discussion of Snort, intrusion detection can be divided into two problems: packet filtering or classification based on header fields and string matching over the packet payload. The first problem was studied extensively in the literature and many algorithms were suggested [35]. A recent study [36] showed that the string matching routines in Snort account for up to 70% of the total execution time. We also have studied the Snort rules and have showed that 87% of the rules contain strings to match against [37]. Therefore, the second problem of string matching is the most computationally intensive. The following section summarizes the academic efforts to accelerate pattern matching by building special hardware.

2.4.2 Intrusion Detection Research in Academia

Academic research continues into new methods of providing IDSs in reconfigurable hardware. The focus is on accelerating pattern matching in hardware which is the most computationally intensive part of IDS. Here we classify the research according to the

technique used to accelerate pattern matching. We discuss the advantages and disadvantages of that technique and list the university groups working in that segment. The main techniques used to accelerate IDS are: regular expressions, discrete comparators, CAMs/DCAMs and hashing. The first four subsections discuss the techniques above, respectively. Subsection 2.4.2.5 presents the research done in the area of stateful IDS. Finally Subsection 2.4.2.6 discusses the most common techniques for behavioral detection.

2.4.2.1 Regular Expressions and Finite Automata

There have been several attempts to accelerate pattern matching in hardware recently, most of the implementations used regular expressions. Regular expressions are a common way to express string matching patterns. The characters to be matched, are the main elements of a regular expression, they are combined with character operators such as concatenation "()" and alternation "|". A question mark matches 0 or 1 occurrences of the pattern in a string, and an asterisk matches 0 or more occurrences of the pattern in a string. For example, "a(b|c)?" matches: "a", "ab", and "ac", the circuit is shown in Figure 2.3 courtesy of [3]. Regular expressions are generated for every string in the rule set and a Nondeterministic/Deterministic Finite Automata (N/DFA) that examines the input one byte at a time is implemented. Java Hardware Description Language (JHDL) was used to generate an HDL circuit description for the regular expressions.

Finite automata machines are complex and hard to implement, the time and space required to build an NFA increases respectively lineally and quadratic with the length of the regular expression. Deterministic finite automata machines, on the other hand, require exponential time and space. Every time a new attack is characterized and a signature is added to the database the FA have to be rebuilt again. Both NFA and DFA can only process on byte every clock cycle resulting in a modest throughput. To achieve higher bandwidth researchers have proposed the use of packet-level parallelism [38], whereby multiple copies of the automata work on different packets at lower rate. As we will discuss in the following subsection, they also suggested the use of wide discrete comparators for the search. Since all comparators work in parallel on the same input (one packet), it is straightforward to increase the processing bandwidth of the system by adding more comparators.

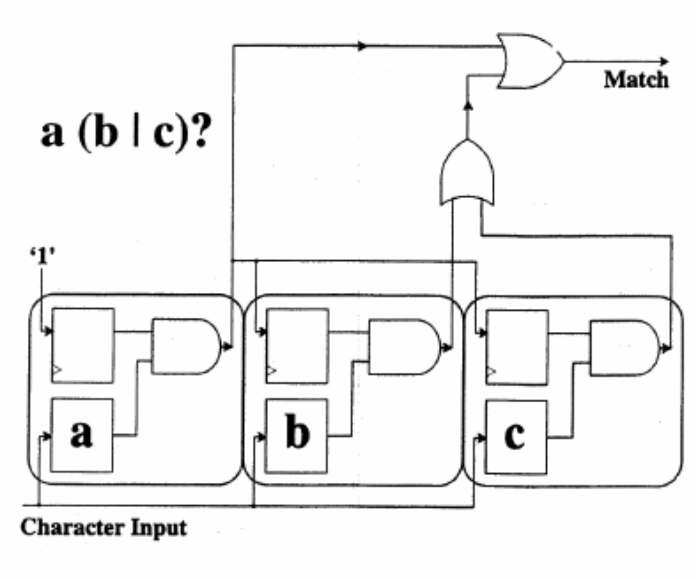


Figure 2.3. Matching “a(b|c)?”

One of the earliest attempts, at the University of Southern California, by Sidhu and Prasanna mapped an NFA into a Virtex XCV100 FPGA [2]. Carver et al., at Brigham Young University, wrote a regular expression generator in JHDL that extracts strings from the Snort database, generates regular expressions and a netlist for a Xilinx FPGA [3]. Lockwood et al., at Washington University, St. Louis, used the Field Programmable Port Extender (FPX) platform to perform string matching using DFAs [38].

2.4.2.2 Discrete Comparators and Pipelining

Due to the fact that if the pattern match operates at one character per cycle, the total throughput is limited by the operating frequency, the attention shifted to using multiple comparators in parallel. Several 32-bit or wider comparators are used in parallel to match every rule against the packet content to achieve higher throughput. Mangione-Smith et al., from the University of California at Los Angeles, for example, used four parallel comparators per rule content [7]. Each rule is translated into structural VHDL unit, which is then synthesized and mapped onto an FPGA as shown in Figure 2.4 (courtesy of [7]). The packet content is passed to the units through a 32-bit bus, i.e. handling four characters every cycle as shown in Figure 2.5 (courtesy of [7]). On the other hand, Sourdis et al., from Technical University of Crete, Greece, utilized N parallel comparators per search rule, so as to process N packet bytes at the same time [6]. They also used extensive fine grain pipelining to achieve higher operating frequencies. The disadvantage of this approach is the

large area required by the comparators and the long latency of the deep pipelines. The other major disadvantage is the area of the comparators where the design needed 3 FPGA devices of 120,000 logic cells to include the entire Snort rule options, and an additional device to accommodate the entire Snort rule header matching.

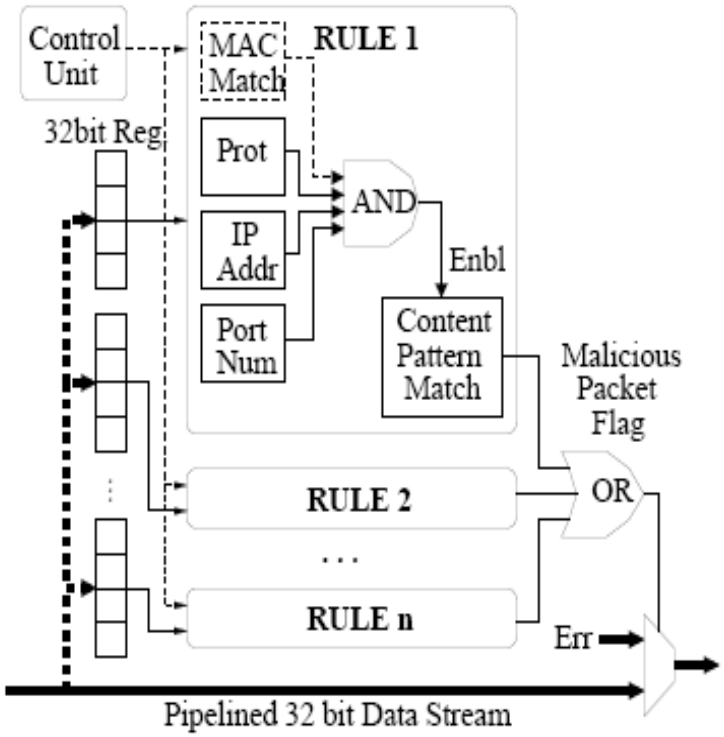


Figure 2.4 Parallel rule units

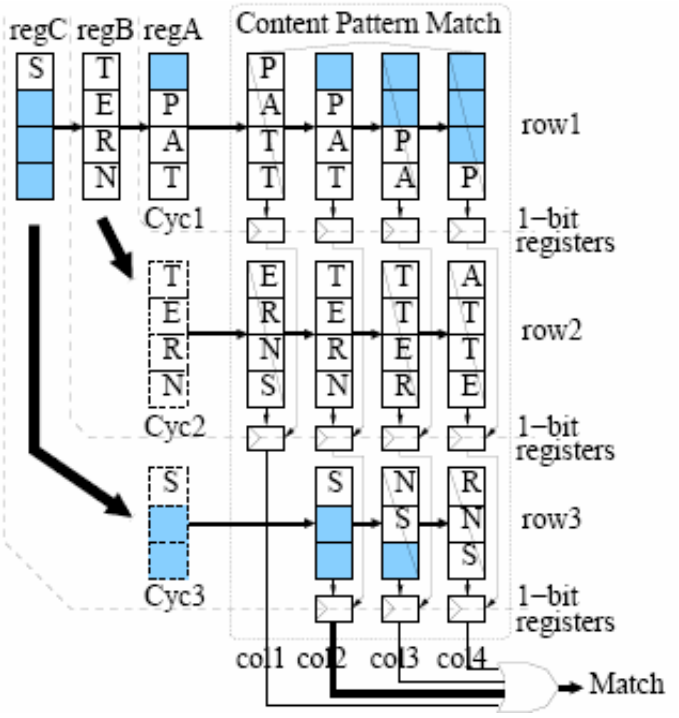


Figure 2.5. Parallel Comparators

2.4.2.3 CAMs and DCAMs

Finite automata machines did manage to increase the performance of pattern matching by about 10 times over the existing software implementations. Despite this ability to run at high speeds, FA-based systems have their own drawbacks. Storing the attack signature in an FA often requires a large amount of resources. As the number of signatures grows from hundreds into thousands or more, on-chip memory becomes insufficient. In order to solve this problem of size limitation while retaining the speed advantage of hardware, pattern matching engines

which use Content Addressable Memories (CAMs) were used. Content addressable memories are initialized with several tables, these include: a protocol table, source and destination IP addresses and content strings. The hardware appends to the packet a match vector that records which CAM entries matched the packet. The rule processor receives each packet scanned by the hardware and correlates the match vector to the internal rule database and determines an action for each rule.

Gokhale et al., from Los Alamos National Laboratory, used a CAM to implement Snort rules on a Virtex XCV1000E [5]. Sourdis et al., extended their discrete comparators IDS with DCAMs [4]. Figure 2.6 shows a CAM based matching unit from Sourdis's design. CAMs based implementations achieve similar throughput to the discrete comparators implementations with reduced area. The drawback is the high cost and the high power requirement of CAMs.

2.4.2.4 Hash Functions

Recently, Dharmapurikar et al., from Lockwood group at Washington University, St. Louis, used bloom filters to perform string matching [39]. The strings are compressed into an m-bit long vector, by calculating multiple hash functions over each string. The hash functions set bits in the m-bit long vector. The compressed m-bit vector is stored into a small memory which is then queried to find out whether a given string belongs to the compressed string set.

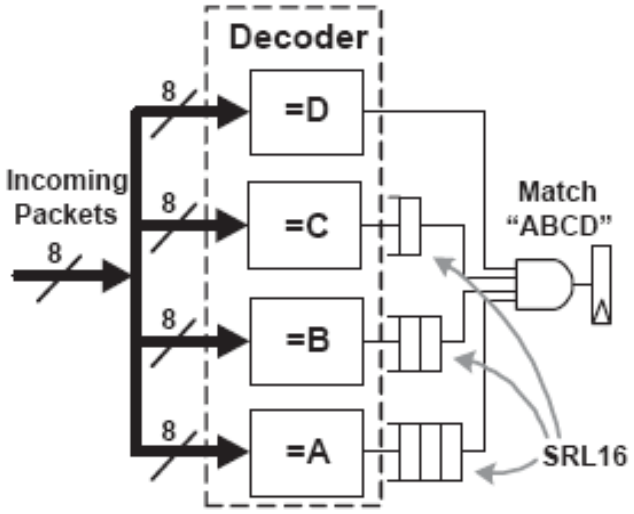


Figure 2.6. Content addressable memories matching

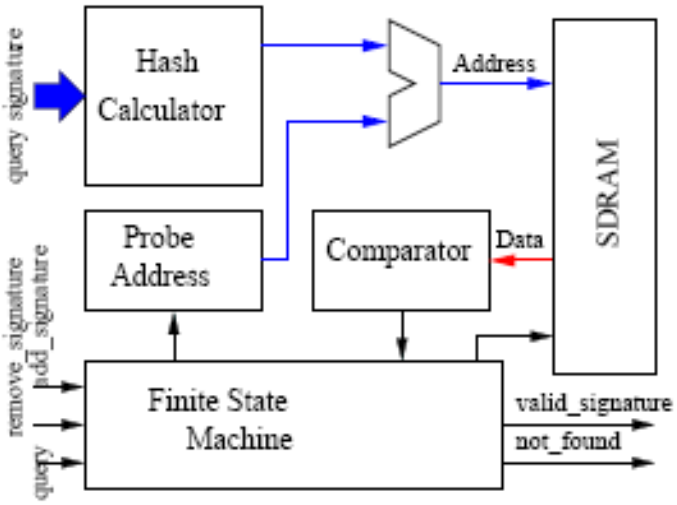


Figure 2.7.Hash table for bloom filters.

The same hash functions are applied to the string, if the resulting vector matches the m-bit long vector then the string is a member of the bloom filter. If a string is found to be a member of a bloom filter, it is declared as a possible match and a hash table or regular matching algorithm is needed to verify the final match. If the resulting vector doesn't match any bloom filter then the string does not match any string in the set and the packet is cleared.

Since multiple strings might set the same bit, it is a little tricky to delete a string from the m-bit vector. Counting bloom filters technique is used, where a counter is kept for every bit in the m-bit long vector representing the number of strings that set each bit. When a string is deleted the corresponding bit counters are decremented. Once the counter reaches zero the bit is reset, meaning that there are no more strings setting that bit. Figure 2.7 shows a block diagram of a bloom filter implementation, courtesy of [39].

The larger the m-bit vector and the smaller the number of strings, the better the bloom filter represents the string set and the better the exclusion queries work. The number of hash functions is a factor of how well the bloom filter works, the more the better. The large number of hash functions and the large vector size can get expensive in some implementations. Compared to the other techniques, bloom filters use less memory, are easy to reprogram and achieve higher throughput than DFA implementations. They eliminate the need to use the expensive pattern matching algorithm unless a membership query is positive.

2.4.2.5 Stateful IDS

A Stateful inspection IDS is able to detect attacks spread over several packets. Once a connection has been established, it is recorded in a table. This table is checked first when packets arrive at the IDS, and if a packet matches the information there, it is allowed to pass. Deep stateful intrusion detection system also examines the packet content up to the application layer and can handle complex connections, such as FTP or Oracle. One of the most useful security benefits of stateful inspection is the ability to enforce protocol standards. For example, SSH creates encrypted connections, therefore many firewalls simply examine the destination port to determine that the connection is SSH. Since it is an encrypted connection, it is assumed that there is little else to be done. In other words, an intruder can get malicious code or packets through the firewall by disguising them as an SSH connection. A stateful inspection IDS examines the first few packets and ensures that they conform to the SSH standard, in other words, that the connection is really SSH, and not something else.

S. Li et al., from the University of Oslo, built modules for TCP reassembly and connection state maintenance in an FPGA [40]. They used Snort misuse IDS and discrete comparators for pattern matching. C. Kruegel et al., from the Reliable Software Group at University California, Santa Barbara, proposed to partition the incoming traffic into smaller portions that can be statefully inspected by IDSs [41]. Intrusion detection is performed by a set of signature based sensors, each of which is responsible for the detection of a subset of the signatures. The system partitions the analyzed event stream into slices of manageable

size. Each traffic slice is analyzed by a subset of the intrusion detection sensors. The traffic partitioning is done so that a single slice contains all the evidence necessary to detect a specific attack, making sensor to sensor interactions unnecessary. G. Vigna from the same group introduced a software system for stateful inspection in web servers [42]. They introduced WebSTAT which is an extension to the STAT framework. State Transition Analysis Technique (STAT) is state/transition-based attack description language designed to support intrusion detection [43].

2.4.2.6 Behavioral IDS

Anomaly detection is based on searching for discrepancies from the models of normal behavior. Anomaly detection has the ability to detect new attacks against computer systems. Application behavior profiles have recently emerged as a substitute for the traditional user based behavior. Application behavior profiles are built by capturing system calls made by the program under analysis, under normal operational conditions. The profile can be used to detect deviations from normal behavior such as those that occur when an application is being misused by an intruder.

One of the first groups to develop application-based intrusion detection was S. Forrest's research group out of the University of New Mexico. They constructed a profile of normal behavior from short sequences of the application system calls (called N-gram) [44]. The

application is monitored under normal usage conditions and a database of system calls is built. If a behavior during the online operation of the application does not match a profile in the normal database, a mismatch is recorded (called the equality approach). If the number of mismatches detected is a significant percentage of all strings captured during the online session, then an intrusion is registered. Subsequent work performed by A. Kosoresow et al., from the University of New Mexico [45], found recurrent patterns of system calls of any given application. They constructed a DFA, by hand, to represent this behavior. DFA based techniques will not scale well to real systems, as the presentation of the program behavior could lead to a state explosion problem. Later work by A. Ghosh, of the Reliable Software Technologies Corp., investigated machine learning such as neural networks back propagation to be able to generalize from previously seen behaviors to map future unseen behaviors [46, 47].

One clear draw back of anomaly detection is its inability to identify the specific type of attack that is occurring. However, the most significant disadvantage of anomaly detection approaches is the high rates of false alarms. Because any significant deviation from the baseline can be flagged as an intrusion, it is likely that normal behavior that falls outside the normal profile captured will also be labeled as an intrusion. Another drawback of anomaly detection approaches is that if an attack occurs during the training period for establishing the baseline data, then this intrusive behavior will be established as part of the normal baseline.

2.5 State of the Art Summary

Network security is a dynamic field where new attacks are catching the headlines every year. There is a race between the attackers, who try to find loopholes, and the vendors, who develop patches for them. None of these attacks is stopped by encryption and not all of them are stopped by firewalls. Intrusion detection systems are becoming more important in the field of network processing where the demand on speed and flexibility is increasing. Most commercially available IDSs are software-based and those systems are not able to keep up with increasing network speeds.

Several approaches to the design of intrusion detection systems currently exist. All of these approaches provide some means for efficiently handling high speed network traffic or adapting to new attacks. The current design techniques used to build these intrusion detection systems do not, however, fully address both aspects of the problem. Unfortunately, they failed to meet both needs of flexibility and speed. For example, a DFA mapped on an FPGA achieve low throughput, they are complex to build and configure. On the other hand, discrete or parallel comparators were used to achieve higher throughput at the expense of increased area and poor scalability. CAM based solutions reduce the area used by discrete comparators and achieve similar throughput, with draw back of increased cost and power consumption. Finally, bloom filters and hash functions were used to compress the string set, find probable matches and reduce the total number of comparisons. Toward this end, a new scheme is

needed to maintain the flexibility of software IDSs while providing the required speed for inline inspection.

Chapter 3

THE ARCHITECTURE

Several approaches to intrusion detection acceleration were described in Section 2.4.2. Although each design had something to offer, it was shown that none of them provided sufficient speed to fully protect network systems at line rate, nor provided enough flexibility to adapt to the ever morphing attacks. In this chapter, we present a new approach to intrusion detection systems design that provides for both increased speed and enhanced configurability. This design is a part of a larger configurable network processor that provides the flexibility and performance required of the network processor while providing an enhanced level of network security.

The rest of this chapter is organized as follows. The first section presents an overview of the network processor components. Section 3.2 discusses the software, complier, algorithms and the state table format. Section 3.3 describes the accelerator hardware, while in Section 3.4 we present the hardware-software interface. Finally, Section 3.5 discusses some of the design considerations and tradeoffs.

3.1 Network Processor Architecture

The configurable network processor architecture shown in Figure 3.1, it consists of a Clustered Length Architecture Word processor (CLAW), a memory, Direct Memory Access (DMA) and a number (*N*) of configurable hardware accelerators. CLAW is a 32-bit load-store processor with Harvard microarchitecture, 5-stage pipeline and virtual memory support. It is a 2-wide multiple issue VLIW machine with hardware support for eight hyper threads. CLAW evolved from the OpenCores.org OR1200 implementation of the OpenRISC architecture [48]. This architecture targets medium to high performance networking, embedded, automotive and portable computer environments. CLAW was written in Verilog, simulated using Modelsim and synthesized using Cadence to program onto an FPGA. GCC compiler version 3.4.2 was ported for CLAW and instruction scheduling is done completely in software. The Verilog code was made available to the open source community [49].

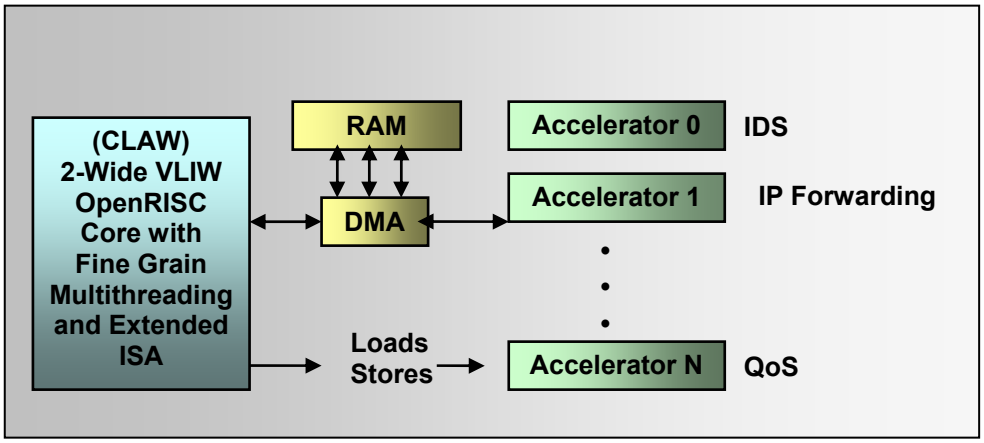


Figure 3.1. Configurable network processor architecture

The memory system consists of multi-port RAM and a high speed Direct Memory Access. The RAM will hold the packets, which then will be streamed to the accelerators. The RAM will also hold the IDS configuration, IP forwarding, Quality of Service (QoS) and classification tables. The RAM is a key component of the network processor and will be the limiting factor in terms of performance and the limiting factor in terms of area and power consumption. The DMA is supposed to provide the accelerators with high speed data access through to the RAM. It will also be responsible for arbitrating the accelerators access to the shared RAM areas.

The configurable accelerators are used to speed up specific networking tasks such as encryption, IP forwarding, quality of service (QoS) and string matching for intrusion detection. CLAW is the core of the network processor and will run any general purpose software as well as the software for the networking tasks mentioned earlier. The software will use the acceleration hardware to speed up certain tasks that are computationally extensive and cannot be executed efficiently on CLAW or any general purpose processor. Pattern matching is the most computationally expensive task in intrusion detection. Therefore, building a hardware accelerator will significantly improve the performance of an intrusion detection system.

In this document we will focus on the IDS accelerator. The accelerator is composed of two components: software that runs on CLAW and hardware for string matching. The software analyzes the rule database and initializes the accelerator. It extracts the strings from

the Snort database, creates the FSM tree and generates the state tables. The hardware is a straight forward FSM implementation that has a RAM to store the state tables and a control logic that traverses the state tables and match the packet against the signature database.

3.2 The Software

The software, also referred to as the complier, converts the raw strings set into an AC tree-based state machine. After the strings are extracted from the Snort rules set, they are in either ASCII format or binary codes represented in hexadecimal characters. The hexadecimal strings are converted into ASCII codes, the spaces are eliminated and the resulting string is used in creating the AC tree. The parser does not support all Snort keywords except for *content* and *uricontent* keywords. The *uricontent* keyword is similar to the *content* keyword except that it is used to look for a string only in the URI part of a packet. The *offset*, *depth*, and *distance* keywords are used to confine the search into a certain range within the packet payload. Those keywords are not supported by the complier. Finally, the pattern search is always case sensitive and the *nocase* keyword is ignored by the complier.

The AC algorithm works by building a tree-based state machine from the set of strings to be matched. The tree is composed from an idle state, child and sibling nodes as shown in Figure 3.2. Every node has a child, sibling and failure pointers for easier traversal and search.

A child node is one that has a prefix matching to its parent node. If there is more than one child, the other siblings are connected to the first child using the sibling pointer.

After the complier is done parsing the strings and converting them into ASCII, the AC state machine is built as described in the pseudo-code in Figure 3.3. Starting with a default no match state as the root node, to add a new string to the tree, the tree is traversed to find the node or child with the longest matching prefix to the new string. If the longest match does not exist, each character of the new string adds a node to the machine starting from the idle state. If a matching prefix was found the rest of the string characters add new nodes to the longest match node. After all the strings are added, the tree is traversed and failure links that point to the longest partial match are added. The algorithm is then optimized by following the failure links to compute the next state for every character from every state in the machine. This shaves a cycle of the clock every time a no-match occurs and a failure link has to be traversed. Rather than having an additional access to the RAM and state tables every failure, the state will be pre-computed by the complier. This adds complexity to the complier in order to be able to achieve higher pattern matching speeds.

After the software creates an AC tree-based state machine, the state machine is then traversed and a state table is generated. The pseudo-code for the traversal algorithm that generates the table is shown in Figure 3.4. Recursion is used to traverse the data structure and fill in the state table. Recursion is very expensive when done in software. However, in

hardware recursion is reduced to a RAM read. The state table is written to the RAM during the initialization of the accelerator.

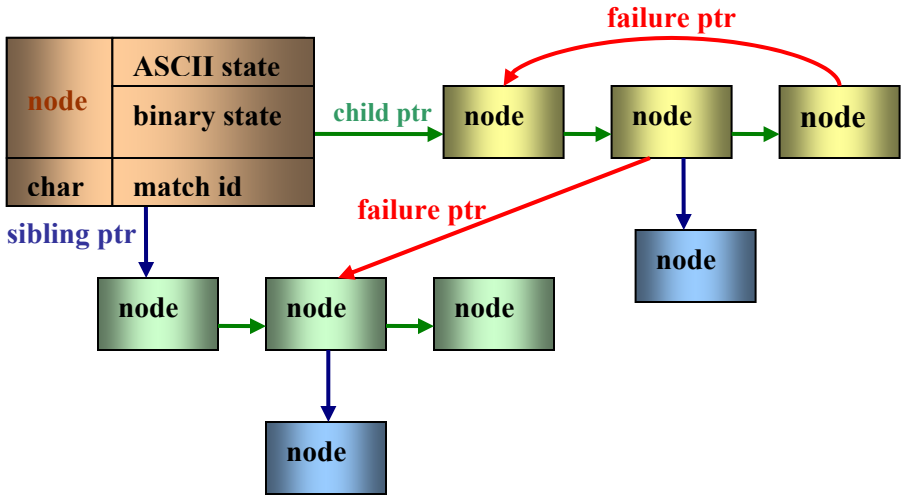


Figure 3.2. General Aho-Corasick tree

```
add string()
        state = idle state
        for every character in the new string
                find the longest matching substring in the tree
                if part of the string exists in the tree
                        add the rest of the string to the tree
                        assign id to the string
                else
                        start from idle and add a new state for every character
                        assign id to the string
        end for
        traverse the tree
                create failure links to the longest matching prefix
                compute the next state based on the failure links
        traverse the tree
        For the characters that cause transition
                print out the final state table
end
```

Figure 3.3. Optimized Aho-Corasick algorithm pseudo-code

```
populate(root)
        populateRAM(root)
        if (root->children)
                populate(root->children)
        if (root->sibling)
                populate(root->sibling)
end


populateRAM( root)
        if (root->children)
        {
                state_table[root->binaryState][root->children->ch)] =
                        root->children->binaryState
                if (root->children->sibling)
                        populateRAM (root->children->sibling)
        }
end
```

Figure 3.4. Pseudo-code for generating the binary state table

Table 3.1 State table

Current State \ Input Character	e	h	r	s	t
- /(idle)	-,0	h,0	-,0	s,0	t,0
h	he,0	h,0	-,0	s,0	t,0
he	-,0	h,0	her,0	s,0	t,0
her	-,0	h,0	-,0	hers,1	t,0
hers	-,0	sh,0	-,0	s,0	t,0
s	-,0	sh,0	-,0	s,0	t,0
sh	she,2	h,0	-,0	s,0	t,0
she	-,0	h,0	her,0	s,0	t,0
t	-,0	th,0	-,0	s,0	t,0
th	the,3	h,0	-,0	s,0	t,0
the	-,0	h,0	ther,0	s,0	t,0
ther	there,4	h,0	-,0	hers,1	t,0
there	-,0	h,0	-,0	s,0	t,0

The state table for the string set and state machine discussed in the previous section (in Figure 2.1) is shown in Table 3.1. The table is simply a translation of the state machine into a tabular format. The rows are indexed by the current state and the columns by the input character. Every element contains a pair of values: the next state and a matching string Identification Number (match id). Only the columns corresponding to characters that cause a transition to a valid state are shown in the table below for space limitations. All other characters will lead back to the idle state.

To match against a packet payload, the system processes the payload character by character starting from the idle state. For every character in the packet, the table is indexed by the duple {current state, input character}. If no entry exists in the table, next state becomes idle and match id is reset to zero. If an entry exists in the table, the table entry for

next state becomes the current state and the match id is checked to see if we matched a string. If no strings were matched then repeat the process until we reach the last character in the packet payload, in that case the packet is clear and allowed through the system. If a string was matched we exit, label the packet dirty and interrupt the processor. The software running on CLAW takes the action specified in the original Snort rule header. The pseudo-code for the matching algorithm is shown in Figure 3.5.

```
match ()
        current state=idle
        match id=0
        string= get (input packet) as ASCII
        for (i=1; i <= strlen(string); i++)
                if (match id) break
                if table[current state, string(i)]
                        {current state, match id}= table[current state, string(i)]
                else
                        {current state, match id}= {idle,0}
        end for
        return match id
end
```

Figure 3.5. Matching algorithm pseudo-code

For example, let's assume the packet payload in ASCII reads "go there". We start traversing the table by resetting the current state to idle and the match id to zero. The packet is processed one character at a time, starting with the letter "g". The table is indexed with the duple [idle, g], since there is no column for "g" then the current state remains as idle. The next two characters are "o" and space " ", there are no table entries for them either. The current state remains idle and match id is still zero. The next duple is [idle, t] for which the table entry indicates that the next state is "t" and match id is zero. The current state is changed to "t" and the table is indexed with [t, h]. The table is accessed and the current state is set to "th". The table is then indexed by [th, e] and the current state is changed to "the". The match id is equal to three, indicating that the packet matched the string in rule number three. Some intrusion detection systems stop when the first match is found and executes the action specified by that matching rule. The current version of Snort creates an event in the event queue and continues processing the packet. Once the packet is fully processed the event selector determines which event has more priority and takes action. Assuming we will continue processing the packet. The duple [the, r] is next and the current state is changed to "ther". Finally, the last character of the packet is processed and the duple [ther, e] is used to index the table. The entry specifies a next state of "there" and a match id of four. The packet was found to match two different strings that belong to rules three and four.

3.3 The Hardware

The string matching accelerator hardware is shown in Figure 3.6 and it consists of an interface, a parser and a matching module. The interface reads the packet from the main memory a byte at a time and will be discussed in further details in the following sections. The parser separates the header fields from the packet content and passes the content to the matching module. The matching module traverses the RAM based AC state tables until it finds a match. The matching module is also responsible for updating the state tables when prompted by the intrusion detection software running on CLAW. One more component not shown in the figure is the slicing module, which is responsible for dividing the input packets into multiple independent packet streams. The Slicing module routes the packet streams to the appropriate matching module. Multiple matching modules are instantiated for the different rule classes to increase the throughput. There will be more information on the slicing module, packet classification and load balancing in the following chapter.

The matching module shown in Figure 3.7 implements a Mealy FSM and consists of a RAM to store the state tables, a register to hold the current state, and control logic to access the RAM, traverse the tree and find a match. The state tables generated by the complier are written directly to the RAM. The control logic is responsible for initializing and updating the state tables. The control logic is a simple FSM that matches the packet content against the string set a byte at a time. The FSM generates the RAM address, reads the next state and match id, and exits if match id doesn't equal to zero (i.e. a match is found). If match id is

zero the FSM processes the next input byte until the end of packet payload or a match is reached.

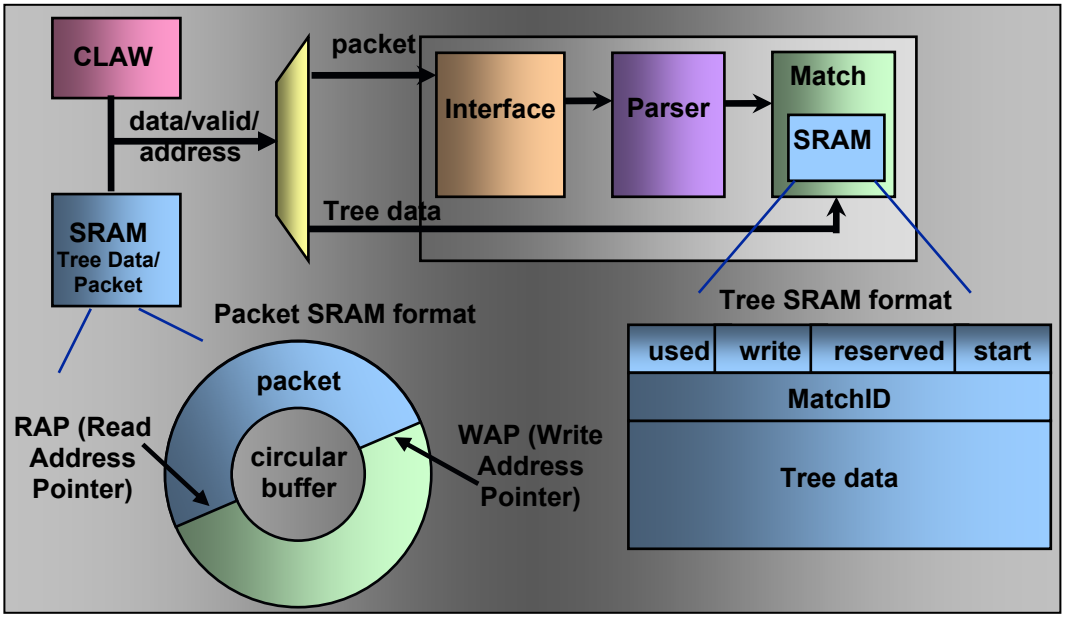


Figure 3.6. String matching accelerator

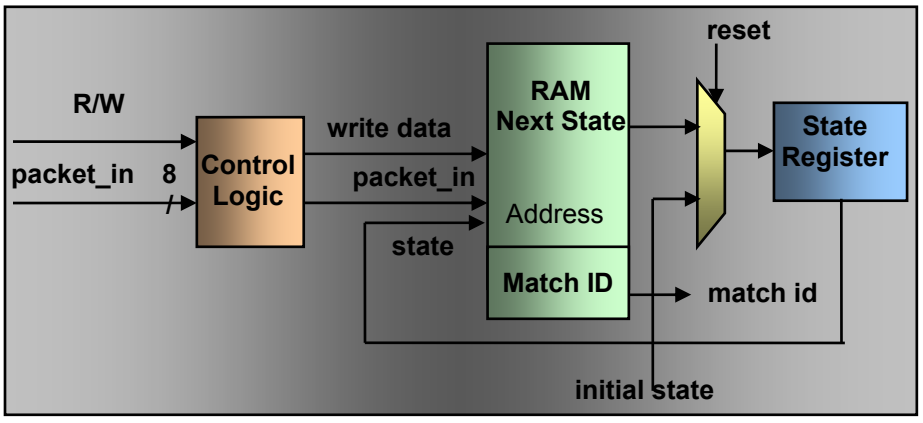


Figure 3.7. Matching hardware

3.4 The Hardware-Software Interface

The processor does not need any special instructions to access the accelerators, because it deals with them as memory mapped peripherals. The high level software simply uses loads and stores to initialize and start the accelerators. The results of the execution (match id) are stored in the local RAM. At the end of execution, the processor is notified through interrupts to read the match id.

The interface is composed of two SRAMs, the packet SRAM and the local tree SRAM. The memory map for both SRAMs is shown in Figure 3.6. The packet SRAM is a circular buffer that stores the incoming packets. It is assumed that the accelerator is much faster than the CLAW processor, that way the buffer acts as a queue of the next data to be processed. The buffer has two pointers that are stored in the first two words of the SRAM. The Read Address Pointer (RAP) is used by the accelerator to point to the next word to read. The Write Address Pointer (WAP) is used by CLAW to point to the last word written. The packet data is stored in between the RAP and WAP. If the RAP catches up with the WAP then there is no packet data to read. If the WAP catches up with the RAP, although that is highly unlikely, then the buffer is full and the processor holds writing until the RAP valued is changed by the accelerator. The use of circular buffers simplifies the interface hardware, in the sense that the processor and accelerator do not need to be able to communicate outside the buffer. The size of the buffer is a design parameter which depends on the average packet size and how fast

the accelerator compared to the processor or network interface card. Circular buffers are generally small and are composed of register banks for increased interface speeds.

The tree data SRAM holds the state tables. The first two words of the tree data SRAM are special words used to control the accelerator and arbitrate the SRAM accesses. The first word has three bits to control access to the tree data. The write bit indicates that CLAW is modifying the tree data, when the write bit is set the accelerator cannot use the tree data. The used bit indicates that the SRAM is being read by the accelerator. CLAW cannot update the tree data if the used bit is set. The start bit is set by CLAW to instruct the accelerator to start processing the packet. The write and start bits cannot be set unless the used bit is zero, this is to insure that the coprocessor is not interrupted while processing a packet. The used bit can only be set if the start bit is one. The second word of the SRAM is used to store the resulting match id. Table 3.2 shows all possible combinations for the accelerator configuration bits.

At boot up all bits are reset to zero. During initialization CLAW sets the write bit, writes the tree data and then resets the write bit. CLAW then writes the packet to the circular buffer and asserts the start bit to start the execution. The accelerator sets the used bit and start accessing the tree data. When the accelerator finishes processing the packet it resets the start and used bits and interrupts the processor. The processor reads the match id stored in the second word. The match id is then reported to the software which decides on the action to be taken according to the action table extracted from Snort rules. CLAW then loads the next packet and asserts the start bit and the process repeats. If a new rule is added, the accelerator

continues to run while CLAW rebuilds the FSM. Once the new state tables are ready and the accelerator resets the used and start bits, CLAW sets the write bit and updates the state table. In the mean while the packets are being buffered in the circular buffer until the configuration is done.

Table 3.2 Accelerator configuration bits

Configuration Bits Combination {start, used, write}	Accelerator and Processor Status
000	The accelerator is idle.
001	The processor is updating the state tables, no packets buffered. Indicates initialization phase.
010	The accelerator is processing a packet.
011	Illegal combination.
100	The processor finished writing a packet and is signaling the accelerator to process the packet.
101	The processor updating the state tables while packets are being buffered.
110	The accelerator just started processing a packet.
111	Illegal combination.

This accelerator was designed to have little or no impact on the existing intrusion detection software. The programmer is not required to be aware of the way the processor is interfaced to the accelerator. Accessing the accelerators is as simple as making a remote procedure call (RPC). The compiler should be able to replace the calls with the string matching subroutines provided with the accelerator.

3.5 Design Considerations

New rules need to be added to detect the emerging attacks and old rules might need to be updated or deleted. The IDS accelerator uses a RAM to store the state tables which makes it easy to reconfigure. To add a new string or delete an existing one the state table has to be rebuilt and written to the RAM. Having one FSM for all the strings in the Snort database is inefficient and will decrease the throughput and increase initialization and configuration time. However, incoming packets need to be matched only against a subset of rules that match the packet header in the Snort database. To avoid creating one large complicated FSM for all of the strings in the database, the software performs a simple rule classification resulting in a smaller FSM or state table for every class. Rules are classified based on the header fields, mainly the protocol and port numbers, into classes such as ICMP, FTP, SMTP, Oracle, Web-CGI…etc. Splitting the Snort rules into smaller sets allows for smaller state tables and faster reconfiguration time.

Having multiple FSMs and state tables allows updating one FSM while the intrusion detection system continues to run. Only the table for that class will be rebuilt, in the mean while all packets destined for that class are buffered. The smaller the class is the less disruption will be due to updating the tables. For example, for a small class like FTP the complier takes less than 5 milliseconds at worst to build the state tables. This delay represents about 34 packets that need to be buffered. We used circular buffers which are a perfect method to buffer only a smaller number of packets. The larger the size of the class,

the worse the configuration time and the larger the buffer needed. In case the classification algorithm could not keep the classes small the processor need to buffer the packets in its memory if the circular buffer fills up. We will examine the configuration time in more details later.

Widening the input bus width is another design issue to consider. It will remarkably increase the performance and allow the accelerator to process packets faster. On the other hand, it will complicate the pattern matching algorithm and increase the time to build the FSM. Processing multiple bytes per second, while useful requires multiple streams of packets waiting to be processed. Assuming we are able to use multiple network interfaces, that will put a lot of strain on the CLAW processor to handle all the packets and route them to the appropriate FSM. The Aho_Corasick algorithm uses one byte to determine the next state, changing the algorithm to process multiple bytes will be quite challenging. Instead of every state having 256 possible next states the algorithm will have to deal with 256^n possible next states, where n is the number of bytes to be processed simultaneously. That means that the amount of data that need to be stored per state increases exponentially with the number of bytes to be processed. This explosion of next states proves to be problematic and impractical. It will only be feasible if the size of the FSM is kept rather small, much smaller than the FTP FSM.

Chapter 4

THE CONCEPT AND ALGORITHMS

In Chapter 2 we emphasized that the current software-based intrusion detection systems do not have the speed required to protect computer networks in real-time. The lack of throughput is due to the large size of the database and slow memory access time. Current AC-based software intrusion detection systems generally use huge data structures to represent a node or a state. Those structures require up to 256 pointers for the next state, every pointer is 32 bits. That's in addition to the state information, character representing the node, partial string matched so far, children pointers, as well as failure pointers. The node size is rather huge and exceeds 1024 bytes. Most intrusion detection systems today have over 1000 rules, which results in hundreds of thousands of states and close to 58 MB of data. It is impossible to host such a huge database on-chip, needless to say it is extremely expensive, hard to build and traverse.

In order to address those concerns, we introduced a new network processor architecture with special accelerators to speedup pattern matching. We paired that powerful architecture

with hardware-tailored pattern matching and compression algorithms to reduce the size of the database and achieve higher pattern matching speeds. They key to reduce the size of the database and achieve higher speeds, is to convert the FSM to a tabular format and program it into a RAM which can easily be accessed through special hardware. In addition to that we split the large FSM into several small FSMs that can be accessed in parallel to achieve higher matching speeds as explained in Section 4.1. Section 4.2 explains the packet classification problem in more details and suggests a few load balancing techniques. Finally, Section 4.3 introduces the Aho-Corasick table compression algorithm and proposed hardware.

4.1 FSM Splitting

The improvement on performance achieved by this work is attributed to converting the large complicated FSM into a tabular format which allows faster data retrieval due to reduced size of the signature database and simplified logic. Software implementations of intrusion detection systems as well as hardware intrusion detection systems that used FAs generated huge data structures and FSMs, respectively. Larger databases needed more memory to store the larger nodes and required multiple memory accesses per state transition. Larger FSM requires more time to build, and it takes more time to retrieve the information. By pre-computing the states and storing them in a tabular format, we significantly reduce the size of the memory required to store the Snort database. We designed a simple acceleration hardware that requires only one memory access to determine the next state. Finally, we

reduce the size of the data retrieved in each access to one byte rather than a huge data structure that exceeds one KB. The reason we have such a significant reduction in data size is due to the fact that we only store those elements that belong to an input character, which exists in one of the strings in the rule set. All elements that belong to failure characters pointing back to idle states are not stored. This is equivalent to permanently deleting all the NULL pointers from the data structure in the software implementation. It is equivalent to removing all failure links to the idle state in FA implementations. All elements that belong to a character already in the string set are stored, including those that point to the idle states. In the following section (Section 4.3) we will present a compression algorithm that will further optimize the table be eliminating all failure links to the remaining idle states and states that has single character.

The second major contributor to reducing the memory requirement and increasing throughput is dividing the large, slow and complicated FSM into several small and fast FSMs. Each FSM is responsible for only a small subset of the Snort database. The advantages of this technique are:

1. The resulting FSMs have smaller number of states and occupy less memory space than the original FSM.
2. Smaller FSMs are faster to build and configure, because there are less possible next states per node.

3. Smaller FSMs are faster to traverse, which increases the intrusion detection system throughput.
4. Multiple FSMs can run in parallel to maximize performance and allow online configuration.

By examining the Snort rule set we find out that the incoming packets need to be matched only against a subset of rules that match the packet header in the Snort database. For example, FTP session packets need only to be matched against the FTP rules subset which has 50 strings as opposed to over 1500 strings in the large FSM. The classification algorithm used, is based on sorting the rules by using IP header fields, mainly the protocol and port numbers. Therefore, we have separate FSMs for different protocols and destination IPs which are mutually exclusive and can be run independently from each other. The accelerator hardware has a slicing unit that classifies the incoming packets by port number and route them to the appropriate FSM. ICMP packets are routed to the ICMP FSM that represents the strings contained in the ICMP rules. Separate FSMs are constructed for ICMP, FTP, SMTP, Oracle, Web-CGI…etc.

The memory requirement in bits was derived in terms of the number of states, number of strings and the number of different characters in the string set. Equation (1) shows the memory requirement (RAM), where s is the number of states, n is the number of strings and c is the number of characters per set.

$$RAM = \left(\lceil \log_2 s \rceil + \lceil \log_2 n \rceil\right) * s * c \qquad (1)$$

The RAM size is directly proportional the number of states. The number of states depends on the number of strings and the number of characters per string. Assuming every character in every string adds a new state to the tree, the worst case number of states can be derived in terms of the number of strings and average string length *sl* as follow:

$$s_{wc} = n * sl$$

Substituting worst case values in equation (1) results in equation (2) below.

$$RAM = k_1 * n * \left(\lceil \log_2 n^2 + k_2 \rceil\right) \qquad (2)$$

Where k_1 is constants that depends on the average string length and the number of characters per set and $k_2 = \log_2(sl)$ is a constant the depends on the average string length. The disadvantage is that the memory utilization increases linearly with the number of strings. The slope of the curve depends on the number of strings and average string length or the number of characters per set. The linear growth is tremendous improvement over the

exponential trend in the software and FA based implementations. Section 4.3 will discuss a compression algorithm that will reduce the slope by a factor of the compression ratio.

4.2 Classification Algorithms and Load Balancing

We explained at the beginning of this document that the purpose of this work is to investigate algorithms and architectures to speed up the pattern matching aspect of intrusion detection. Despite the fact that pattern matching is the most computationally extensive task in intrusion detection, we found that by employing a simple classification algorithm we can achieve higher throughput. Classification algorithms are used to exploit parallelism between the different rule classes as well as different packet streams. With splitting FSMs and packet streams, load balancing becomes an issue. An algorithm that classifies the traffic in such a way where only few classes are in use, defies the purpose of improving throughput. In an ideal world, the classes will all have similar number of rules and a fair share of the overall traffic. When choosing a classification algorithm several factors must be considered, including the traffic distribution, the rule distribution and the number of strings in each class.

Packet classification is a separate field and has been studied extensively in literature, because of that we opted out to using a simple classification algorithm based on the port numbers. We classify the incoming packets and the Snort rules by the destination port and port fields, respectively. The resulting Snort rules classes along with the number of

characters are shown in Table 4.1. We notice that the HTTP classes have a very large number of rules, which means that the HTTP FSM will probably by larger and slower. Most of the Internet traffic today consists of HTTP packets, leaving the large slow HTTP FSM extremely busy and the other FSMs slightly under used. Having only a few large classes or a few streams with high traffic creates bottlenecks. An improvement over the existing classification algorithm to mitigate the shortcomings of load imbalance would be to dedicate several FSMs to the larger classes. Several identical FSMs representing the full class will be used in parallel to operate on different parts of the traffic stream. This is not an ideal solution in the sense that it just hides the latency created by large the classes because the individual FSM are still large and slow. A better classification and load balancing algorithm is needed.

Table 4.1 Snort rule classes

Rule class	# of Chars	Rule class	# of Chars	Rule class	# of Chars
ftp	397	chat	409	web-php	1264
imap	162	backdoor	382	web-coldfusion	949
snmp	36	icmp-info	204	web-frontage	629
smtp	355	info-rules	109	web-client	99
scan	115	multimedia	144	rservices	113
dns	238	netbios	239	shellcodes	241
finger	49	oracle	353	policy	172
porn	251	sql	525	attack-reponses	227
pop2	35	p2p	140	mysql	15
pop3	121	rpc	806	x11	19
nntp	18	tftp	30	bad-traffic	0
telnet	130	virus	640	other-ids	46
icmp	138	web-attcks	451	misc	418
dos	65	web-cgi	4727	web-php	1264
ddos	180	web-misc	4235	**Total**	**22143**

Packet classification has been always associated with quality of service, where the main objective is to classify the Internet traffic into classes with different service and bandwidth levels. In the early stages, packet-classification and forwarding was based on simple Layer 2 through Layer 4 look-up of the packet header. Today, network applications such as load balancing, web switching and intrusion detection require deep packet inspection. It is apparent that classification of packets in the Layer 5 through Layer 7 fields (packet payload) is needed. Packets that share common application layer or session layer information are grouped into a separate stream.

As mentioned earlier the Internet traffic is being dominated by web based services. Web-based applications primarily use the Hyper-Text Transfer Protocol (HTTP). An HTTP packet is normally identified by the destination port number (usually 80, 8000 or 8080) that is contained in the header. The data embedded in the HTTP packet can be as simple as browsing a web page or as complicated as Voice over IP (VoIP) and streaming multimedia applications. The large HTTP class discussed earlier can be divided into smaller more efficient classes based on the traffic encapsulated within the HTTP packets. The traffic can be classified based on the application name contained in the Uniform Resource Locator (URL) which is present within the data payload of an HTTP packet. A more advanced content classification involves searching for a cookie field, which is a piece of information exchanged between an HTTP server and a web browser to maintain the application. Because

these fields do not reside in a specific offset within the packet, complex search operations must be performed to locate and classify them.

Because of the number of protocols being used and the fact that there is not single field to effectively classify packets, we propose label classification. The idea is not new and has been used before in ATM networks as well as the Multi-Protocol Label Switching (MPLS) protocol. Label classification is achieved by adding a field to the packet header identifying or labeling the nature of the payload or application. That label is then used by the packet classifier to route the packet to the appropriate destination. The deep labeling classification algorithm will produce finer classes and FSMs with more balanced loads.

4.3 State Table Compression Algorithm

The algorithms described in the previous sections allow for a smaller memory requirement and enable the design to achieve higher throughput. More bandwidth can be achieved by replicating FSMs and running them in parallel. Replicating FSMs has diminishing returns and the number of parallel FSMs is limited by the classification algorithm.

By a thorough examination of state table presented in Table 4.2, we make the following observations:

- Not all the elements carry useful information, in other words states that cause a transition to new and non idle states. There is a significant number of states that point back to the idle state, we call these the failure states.
- There is a significant number of elements that point to states with a single character, we call them the single character states.

We can significantly reduce the size of the state table by storing only those states that cause transition to useful states. We can further reduce the size of the table by compressing the space used for single character states. The failure and single character states are repeated several times through out the table. We came up with a new algorithm and architecture that will effectively store and retrieve the repeated information. The redundant elements are categorized into one of the following classes:

1. Elements with idle as the next state. Those states represent a combination of current state and input character that doesn't have match or have a matching prefix to any of the strings in the set. The green box labeled "1" in Table 4.2 shows an example of such elements.
2. Elements with a next state that has only one character. Those belong to a character that appears only once at the beginning of one or more strings. Box "2" in red shows an example of this class.
3. Elements with a next state that has only on character but belong to a character that appears in any location in one of more strings. Box "3" in blue shows an example of this class.

Elements in classes 2 and 3 belong to characters that don't repeat often in signatures. Special symbols and less frequent letters in the alphabet such as such as §, ¤, j, x, q, z, are an example. A better example would be the hexadecimal combination for unprintable characters represented sometimes on monitors by the symbol "□ ", such as x7F. Those characters might appear in virus signatures as binary codes but are less common than letters of the alphabet. Those observations can be extended to eliminate all common prefixes or all elements that repeat. Those elements are not very common and the time and effort to find a compress those prefixes is more expensive than the savings attained by eliminating them. Therefore, we will not widen our optimization algorithm further than single character states.

Table 4.2 Redundant state information

Current State \ Input Character	e	h	r	s	t
- /(idle)	-,0	h,0	-,0	s,0	t,0
h	he,0	h,0	-,0	s,0	t,0
he	-,0	h,0	her,0	s,0	t,0
her	-,0	h,0	-,0	hers,1	t,0
hers	-,0	sh,0	-,0	s,0	t,0
s	-,0	sh,0	-,0	s,0	t,0
sh	she,2	h,0	-,0	s,0	t,0
she	-,0	h,0	her,0	s,0	t,0
t	-,0	th,0	-,0	s,0	t,0
th	the,3	h,0	-,0	s,0	t,0
the	-,0	h,0	ther,0	s,0	t,0
ther	there,4	h,0	-,0	hers,1	t,0
there	-,0	h,0	-,0	s,0	t,0

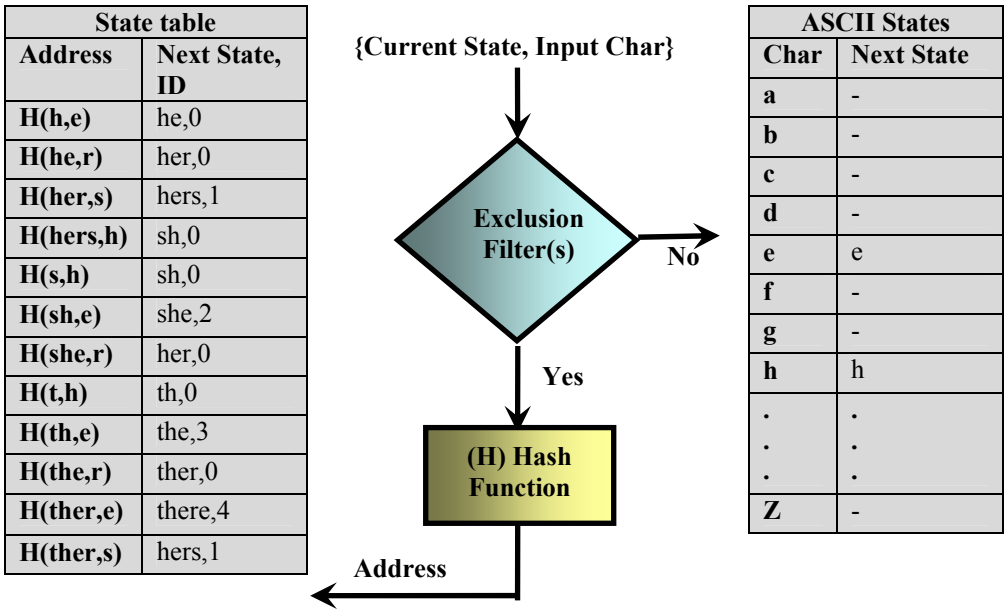


Figure 4.1. AC state table compression algorithm

The hardware for the compression algorithm is similar to that of the accelerator presented in Figure 3.6. The main differences are the way the RAM address is generated and the fact that we have two separate tables for generating the next state. Figure 4.1 illustrates how the compression algorithm can be implemented. The state table will be reduced to only states which have a next state containing more then one character. Since a lot of elements are removed from the table, the addressing scheme used earlier will not results in a continuous space and will have a lot of gaps. Because of the scattered storage space is spread over different addresses a hash function is used to determine the new address. The hash function

converts the current state and input character duple into addresses that belong to a continuous address space. The hash function not only generates contiguous storage space but also allows for faster access to data. The hash function used is a one-to-one function, the actual data items are stored directly at the address given by the hash function. The inputs to the hash function are the current state and input character, the output is normalized against the size of the table. A separate table with 256 entries, or less, is kept for the single character states (ASCII states). The next state is equivalent to the character if it exists in the string set. Otherwise, no entry for a character indicates an idle state. Figure 4.2 shows the compression algorithm pseudo-code. The pseudo-code illustrates how the new tables are generated out of the old state table. The old table is traversed where idle states are ignored and the single character states are stored in the ASCII states table. The rest of the duples are programmed into the bloom filter and added to the compressed state table at the address determined by the hash function.

To match a packet against the compressed table, the packet is processed one character at a time as explained earlier. The exclusion bloom filters are queried to determine if the combination of input character and current state has an entry in the state table. If the filter indicates an entry the hash function computes the address and the state table is accessed to determine the next state. If the exclusion filter determines that the duple doesn't have an entry in the state table, the ASCII states table is accessed to determine the next state. If no entry exists in the ASCII states tables then the next state is reset to idle.

```
compression algorithm ()
        initialize the bloom filter
        for every entry  in the state table accessed by{state, character}
                if (state is idle) do nothing
                if (state is single character state)
                        add to the single character states table
                else
                        address= hash (state, character)
                        compressed table [address] = next state
                        program the duple into the bloom filter
        end for
        return compressed and single character tables
        return bloom filter
end
```

Figure 4.2. Compression algorithm pseudo-code

For the example given in Table 4.2, we can see that the size of the state table was reduced from 13x5x4=260 byes to 12x4=48 bytes plus the table for single character states which has only to contain 3 entries four bytes each. We can see that the savings are enormous (more than 4 times), but that's due to the fact that we have a very small string set. The larger set is and the more diverse in terms of characters the less we can save. In conclusion the

memory requirement still grows linearly with the number of character, but the compression algorithm reduced the slop of the curve by a factor of CR. CR stands for the compression ratio which we will measure in our experiments. Taking the compression ratio into consideration Equation (2) becomes

$$RAM = k_1 * n * CR * \left(\left\lceil \log_2 (n * CR)^2 + k_2 \right\rceil \right) \qquad (3)$$

Chapter 5

THE RESULTS AND ANALYSIS

Several approaches to the design of intrusion detection systems currently exist, each provides some means of improving performance and configurability. Recently, there has been an increased focus on building special purpose hardware that can provide enough power to run IDS in real-time. Most of the techniques currently in use do not, however, provide enough throughput to match the increasing network speeds. While these solutions are steps toward higher speed, they still fall short in configurability.

In this chapter we present the results of our simulations and experiments. The first section presents statistics we obtained from studying the Snort rule set. In Section 5.2 we present the experimental memory requirements. In Section 5.3 we examine the performance of our accelerator. Section 5.4 shows the configuration in terms of compilation time and memory access time. Section 5.5 presents the improved performance and configuration time after applying the compression algorithm. Finally, Section 5.6 compares our design to the to the state of the art hardware accelerators in the academic research.

5.1 Snort Rules Analysis

We studied the Snort rules set released in Oct. of 2003 which contained 1777 rules. Our study shows that 87% of the rules (1542 rules) contain strings to match against the packet payload. That emphasizes the importance of the pattern matching part of the problem especially since pattern matching of strings is more computationally expensive compared to headers classification. This once again, demonstrates the strong need for hardware acceleration of the string matching aspect of the IDS problem. The large number of strings justifies the use of multiple string matching algorithms such as Aho-Corasick used in our design.

The bar chart in Figure 5.1 shows the distribution of the string lengths in bytes. The columns represent the number of rules containing strings of certain length that is marked on the x axis. The columns are clustered around the average string length of 14 bytes. We can see that the majority of the strings are shorter than 26 bytes. It is also clear that there is a non-negligible number of strings longer the 40 bytes. Some of the strings are as long as 184 bytes. Therefore, justifying our decision to use AC as opposed to BM. BM has a run time that is proportional to the length of the rules in the database as well as the number of input characters.

Our simulator took into consideration all ASCII characters including the non-printable characters as well as the hexadecimal strings included in most Snort rules. To make sure that the strings accurately represent the rules they were extracted from. Multiple strings in the same rule within a distance of zero were combined into one string.

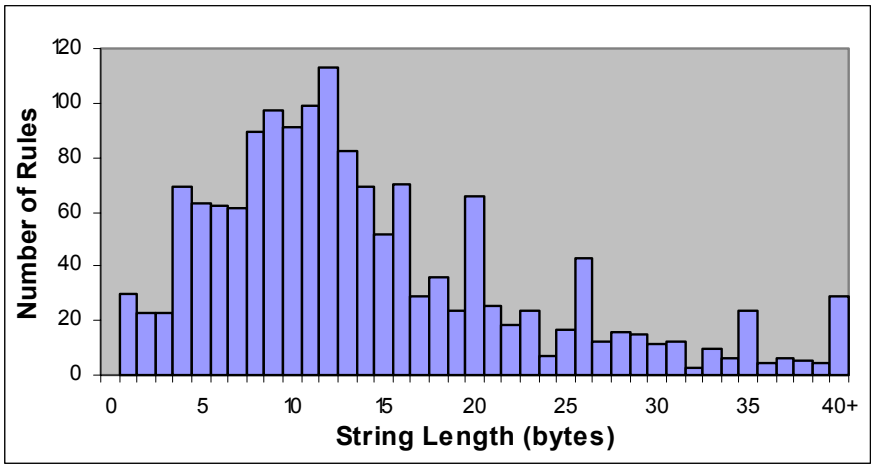


Figure 5.1. Distribution of the string lengths in the Snort database

5.2 Memory Size

The memory is the main component of the accelerator. Hence, it dictates the accelerator size and limits the throughput. Every time a packet is checked and before it cleared, the RAM will be referenced once for every byte. This accounts for the most of the run time of the accelerator. We plot the memory requirements in KBs against the number of characters in a string set. As shown in Figure 5.2 the memory requirement increases linearly with the

number of characters in the string set. This is quite an improvement over the exponential growth of memory requirements of FA-based implementations. This is consistent with the theoretical analysis for the memory requirements presented by equations (1) and (2).

As mentioned earlier, the rules were classified by headers and a separate FSM was created for every class. Since the number of states depends on the number of strings and the number of characters per string, dividing the string set into smaller FSMs will greatly reduce the memory requirement. Table 5.1 shows the RAM requirements for the different rule classes. For space limitations only a subset of the classes is shown in the table, the rest of the classes were summed under the category “other classes”. We notice that we have classes with up to 300 rules and others with rule set as smalls 18 rules. The number of states is greatly reduced with the number of strings which in turn lessen the memory requirement. The RAM requirement for the largest rules classes (web-cgi, web-misc) is around 750KB. The size of the state tables for all of Snort 2003 rule set is around 3MB which can be fitted on-chip. This is down from 58MB required by Snort to store the same rules set. We managed to compress the state table by more than 19 times. By applying the compression algorithm discussed in the previous chapter the size of the state table is reduced even more.

Table 5.1. RAM size in bytes for different rule classes

Rule class	Rules	Strings	States	RAM (bytes)
FTP	50	49	268	43,997
SMTP	18	24	362	56,840
ICMP	22	11	138	17,501
RPC	124	58	720	132,623
Oracle	25	25	265	40,366
Web-CGI	311	311	3133	747,939
Web-Misc	275	275	3242	768,975
Web-IIS	108	108	1514	314,652
Web-PHP	58	58	914	172,132
Web-Coldfusion	35	35	572	98,081
Web-Frontpage	34	34	367	59,926
Other classes	717	554	-	709,963
Total	1777	1542	-	3,118,996

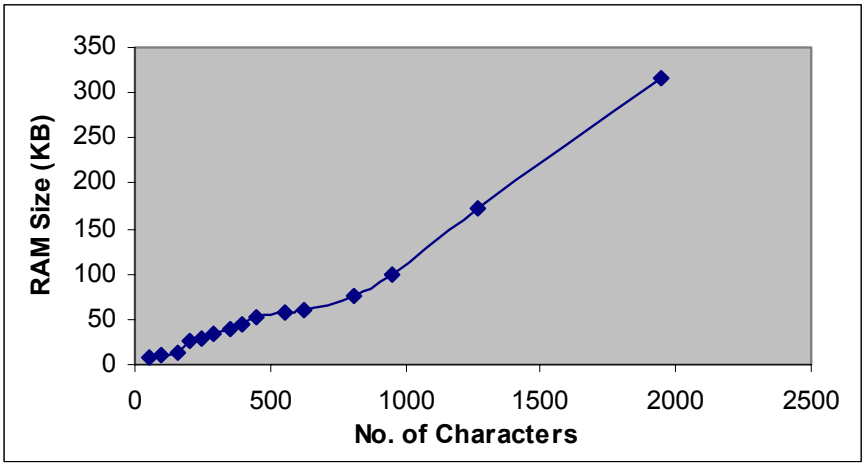


Figure 5.2. RAM size in KB for different character counts

5.3 Performance

The Aho-Corasick string matching algorithm has a deterministic worst-case lookup time. Once the state tables are generated and stored in the RAM, the packet is processed one byte at a time, and every byte requires one access to the RAM. The processing time mainly depends on the length of the packet and the RAM access time. The RAM access time depends on the RAM size which is proportional to the number of strings and the number of characters per string. As discussed the previous chapter, using a simple classification technique to divide the rules into smaller rule sets generates separate FSMs that can run in parallel. This not only significantly reduces the size of the state tables but also increases the throughput by exploiting parallelism between the packets and different rule classes.

Figure 5.3 shows the throughput of the accelerator where the CACTI versions 3.2 and 4.2 [50,51] were used to model the on-chip RAM. The CACTI is an integrated cache access time, cycle time, area, and dynamic power model. The tool allows setting the feature size, input line width and total size of the RAM. The performance numbers below are based only on the RAM performance and exclude the logic delays. The RAM access time were measured and used to determine the throughput. A feature size of 0.35 micron was used to represent the stat of the art FPGA memory blocks. The figure plots the performance in terms of processing throughput (Gbps) for different FSM counts (1, 4, 8) and for rule sets with sizes up to 3,000 characters. By looking at only one curve, the blue curve corresponding to four parallel FSMs, we notice that the throughput is decreasing almost linearly as the number

of characters increases and then flattens out. The knee after which the curve flattens out is close to 1000 characters mark. Therefore, it is important to keep the number of characters per FSM below that point to achieve the maximum performance. The smaller the better, but there is also a diminishing return for having too many FSMs, that's caused by the overhead to route the incoming packets to the appropriate FSM. This will also make the control logic more complicated and the crossbar slower. A thousand characters correspond to around 72 strings per FSM, assuming the string average length of 14 bytes we found from out study of the rules.

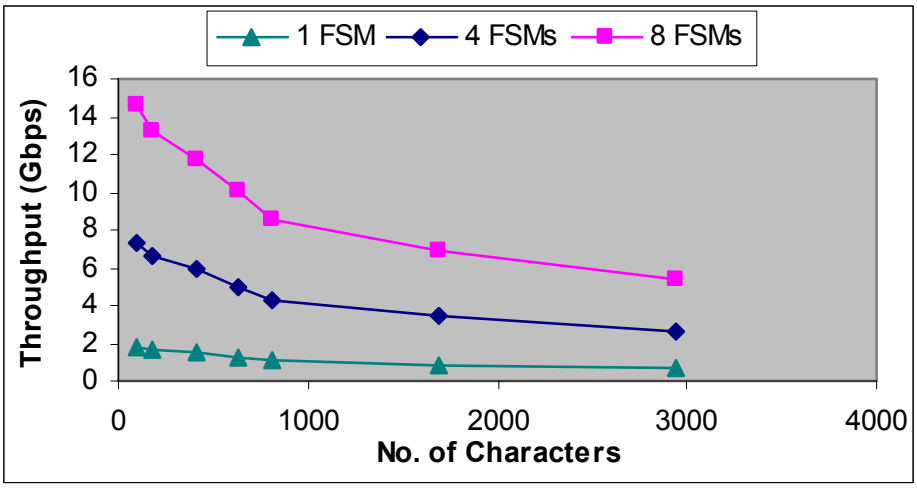


Figure 5.3. Throughput for different character counts

By examining the three curves, we see that higher throughput is achieved by using more FSMs in parallel. By using 8 FSMs a throughput of around 9Gbps is achieved as opposed to

5 and 1.5Gbps for 4 and 1 FSM(s), respectively. We also notice that the throughput degrades as the number of characters grows. The throughput for 8 parallel FSMs decreases to about 5Gbps for 3000 characters. The performance degradation is due to the fact that as the number of characters increases, the number of states increases and the state table size increases as well.

Today some companies at the leading edge of technology are working on a feature size as small as 0.05 micron. If we use a high speed SRAM with a feature size of 0.13 micron the access time of the RAM is greatly reduced and the throughput is boosted to about 20 Gbps for 4 FSMs in parallel with an 800 character class. Figure 5.4 shows the throughput versus the number of character for the new high speed RAM.

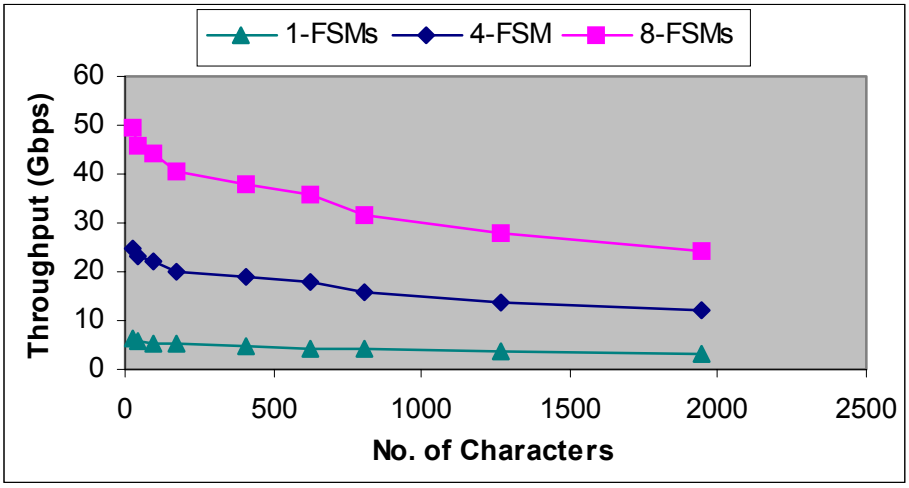


Figure 5.4. Throughput for 0.13µ RAM

5.4 Configuration Time

Configuration time is the time to run the complier, generate the state tables and initialize the SRAM. The time depends on one of the following factors:

1. The general purpose processor running the compiler software. This depends on the processor clock rate as well as primary and secondary memories speed and size.
2. The details of the AC algorithm implementation. We used a linked list similar to the one used in Snort. It has child and sibling pointers to make the navigation faster. Our optimized AC adds one more tree traversal to pre-compute failure states. On the other hand, pre-computing the next state reduces the number of memory accesses for matching a string. The memory is allocated and freed dynamically to save processor resources.
3. The number of strings being processed. The more strings the larger the FSM and the more time it takes to build. We implemented a classification technique to insure a small number of strings per set.
4. Time to access and update the RAM. This depends on the accelerator hardware and memory technology selected.

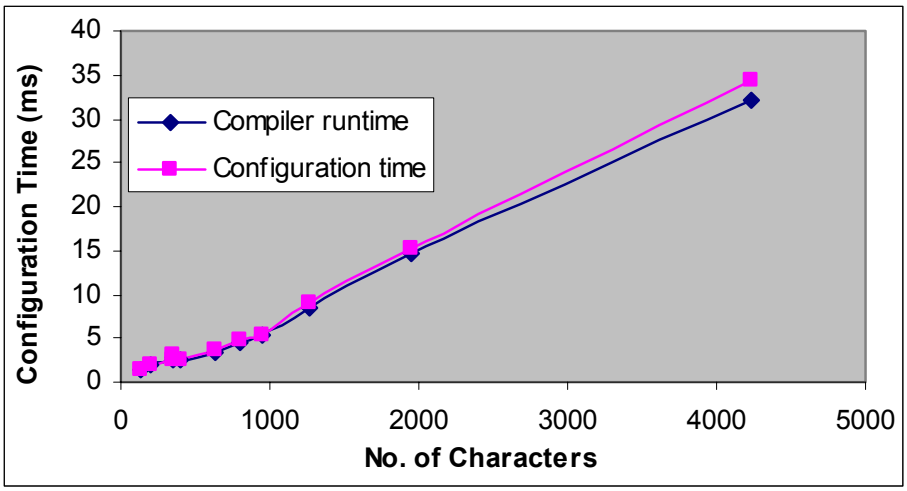


Figure 5.5. Configuration Time

Figure 5.5 plots the compiler runtime and the total configuration time versus the number of characters in a string set. The time is increasing linearly with the number of characters. The two curves run almost over each other and start to diverge as the number of characters increase. The difference between the two curves is the RAM write-time, which explains why the curves move away. As the number of characters increase, the RAM size grows and the time to initialize it becomes greater. Even at 4000 characters the complier time is still the dominant component, this shows the difference between the software running on a general purpose processor and a hardware component.

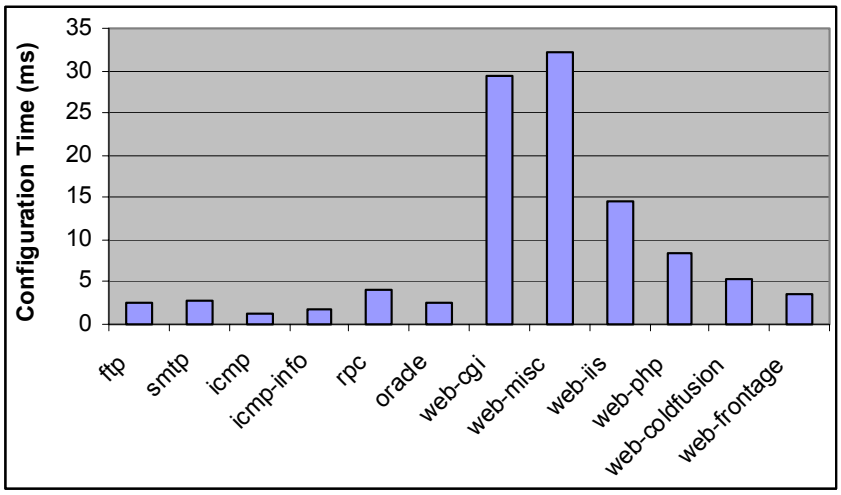


Figure 5.6. Configuration time per class

Figure 5.6 shows the complier runtime for the different rules classes discussed earlier. Smaller classes have very small configuration time compared to the large classes that contain a larger number of rules. Small classes such as ICMP and FTP, with rules sets of 22 and 50 strings, have a configuration time of 2.5 ms and 2.9 ms respectively. While large classes such as Web-CGI, with 311 rules, has a high configuration time of about 29 ms. The overall time is in the order of milliseconds and is very acceptable. Similar hardware implementation has configuration time in the order of seconds, exposing the system to attacks during the long configuration time. Our system supports online updating, that is the system continues to run while only the FSM being updated is brought down. The packets destined to that class are buffered while all other packets continue to be checked. Buffering the packets will not be very expensive given the fast configuration time and small classes. For example, for a small class like FTP the complier takes less than 5 milliseconds at worst to build the state tables. A

leading edge SRAM built using a 0.13 micron feature size will have a total access time of 1.5 ns. With the FTP database size slightly less than 131 KB, the total configuration time will be 5.2 milliseconds. Assuming a worst case hypothetical scenario of a 10 Gbps network interface card directly connected to the FTP accelerator, the 5.2 seconds delay accounts for 34 packets to be buffered. The larger the size of the class, the worse the configuration time and the larger the buffer needed.

5.5 Compression Algorithm Analysis

The compression algorithm works by storing only those states that cause transition to new states that have multiple characters. The algorithm eliminates the space needed to store all elements that point back to the idle state or the single character states. Finally, the algorithm reduces the memory requirements and enables the design to achieve higher throughput.

Table 5.2 shows the RAM requirements for the different rule classes after applying the compression algorithm. Those are the same classes shown in Table 5.1, with column six labeled “RAM CA” showing the memory requirements after applying the algorithm. Column seven labeled “CR%” shows the compression ratio achieved for every class. The compression ratio is the percentage ratio of the new compressed RAM size to the old RAM size before applying the compression algorithm. Therefore, the smaller the compression ratio the smaller the memory we need.

Table 5.2 Compressed RAM size in bytes for different rule classes

Rule class	Rules	Strings	States	RAM (bytes)	RAM CA	CR (%)
FTP	50	49	268	43,997	6,108	13.88
SMTP	18	24	362	56,840	4,975	8.75
ICMP	22	11	138	17,501	1,561	8.92
RPC	124	58	720	132,623	9,964	7.51
Oracle	25	25	265	40,366	3,634	9.00
Web-CGI	311	311	3133	747,939	36,705	4.91
Web-Misc	275	275	3242	768,975	151,968	19.67
Web-IIS	108	108	1514	314,652	49,563	15.75
Web-PHP	58	58	914	172,132	14,843	8.62
Web-Coldfusion	35	35	572	98,081	4,247	4.33
Web-Frontpage	34	34	367	59,926	2,553	4.26
Other classes	717	554	-	709,963	75,406	10.62
Total	1777	1542	-	3,118,996	361,527	11.59

The overall compression ratio is 11.59, meaning that we managed to reduce the total RAM requirements to 11.59% of the original 3.1 MB. That's about close to nine times reduction in memory requirements. The RAM required to store the compressed rule set is 361.5 KB, which can easily fit on-chip. As it can be seen from the table the compression ratio varies from a class to the other. The CR ranges from 4.91% (20 times reduction in RAM size) for Web-CGI class to only 19.67% (5 times) for Web-Misc class. The CA compresses the tables to store only the states that cause transitions to states with multiple characters. This means that string sets with shorter strings or with repeated string prefixes and characters will have fewer combinations that cause transitions and can be compressed more effectively. In general the CR depends on several factors:

1. Number of strings in the set or the number of characters per set. In general larger sets result in larger CR, but this depends on the other factor below. If the set is too small there will be diminishing returns as well and the CR will be large, for example if we have a very small set, say one string, there isn't much room for improvement and CR will be large.
2. String diversity. That is, how different the strings in a set from each other. A string set with common prefixes, has fewer states than another set with similar string count but no common prefixes. Most of the new strings create new states and there is no reuse of common prefix states resulting in large CR. The string diversity is very hard to gauge, it requires a careful study of the string set as well as the AC tree.

The factors above are not mutually exclusive and affect each other. A string set that has short strings is expected to have a small CR unless the strings have a high variation and include all characters in the ASCII code. For example, Web-CGI, Web-Misc and Web-IIS are among the largest rule classes with 311, 275, 108 strings each and 4.7k, 4.2k, 2k characters, respectively. According to first factor above, they all must have a very large compression ratio, which means a very little compression can be achieved. That's true for Web-Misc and Web-IIS with CR of 20 and 16. In other words the Web-Misc and Web-IIS databases can be compressed only by 5 and 6 times, respectively. On the other hand, Web-CGI has a very small compression ration of 5 and the database can be reduced by factor of 20. Although Web-CGI contains more characters than either Web-Misc or Web-IIS, it did

have a much better compression ratio. A simple investigation of the string in the Web-CGI class shows shorter strings with common prefixes such as:

/whois_raw.cgi?|0a|

/whois_raw.cgi

The Web-CGI set has little string diversity and a lot of common prefixes increasing the chance for compression. This highlights another important factor when choosing a classification or splitting algorithm. The algorithm must classify the strings in such a way to maximize the common prefixes, characters and states.

In Figure 5.7 we plot the memory requirements in KBs against the number of characters in a string set, both before and after compression. The pink curve shows the memory requirements after compression. When compared against the blue curve representing the requirements before compression, the reduction in memory requirement is evident. The same linear trend continues to be true, but after compression the curve slope is much smaller as predicted by equation (3). The memory requirement increases at a much lower rate after compression.

Figure 5.8 shows the complier runtime versus the number of characters both before and after compression. The pink curve (configuration time after compression) runs parallel to and above the blue curve (configuration time before compression). We observe that the previous linear trend continues with a slight increase in configuration time. The increase is due to the

complexity added by the compression algorithm, mainly the time to traverse the tree and generate the compressed tables. The worst case time is still in the order of milliseconds, a small price to pay for such a huge improvement in memory space and throughput.

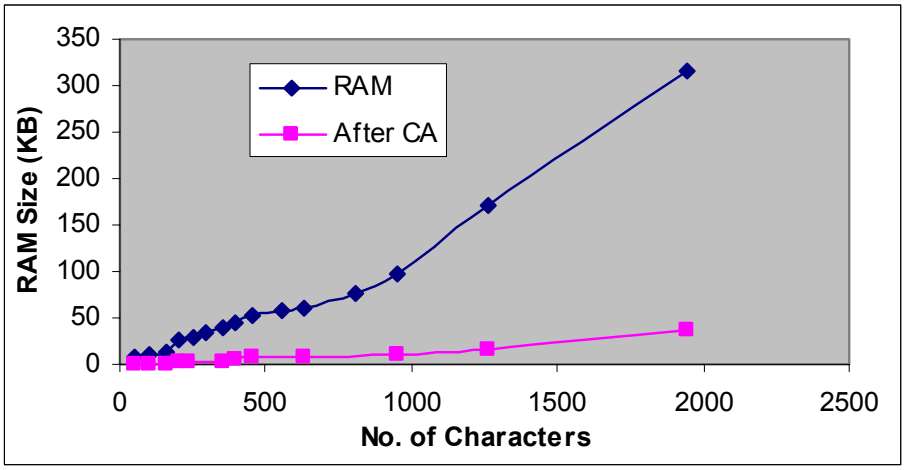


Figure 5.7. Compressed RAM size in bytes

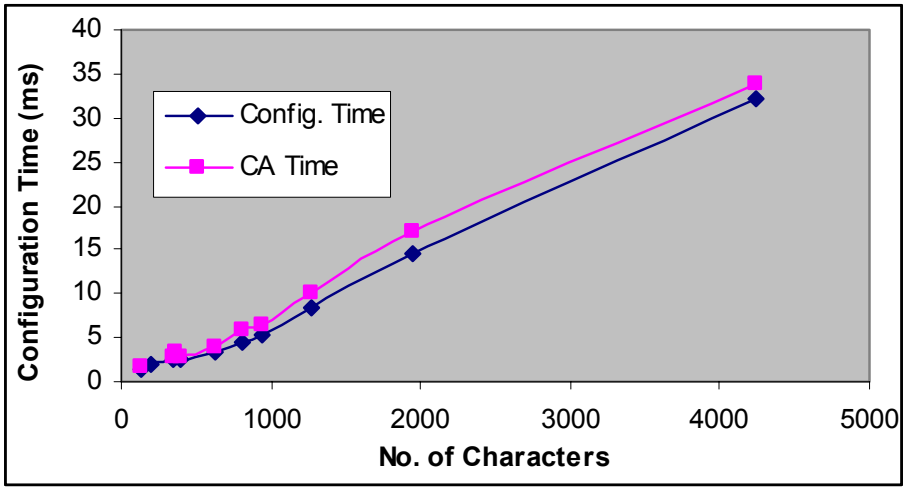


Figure 5.8. Configuration time after compression

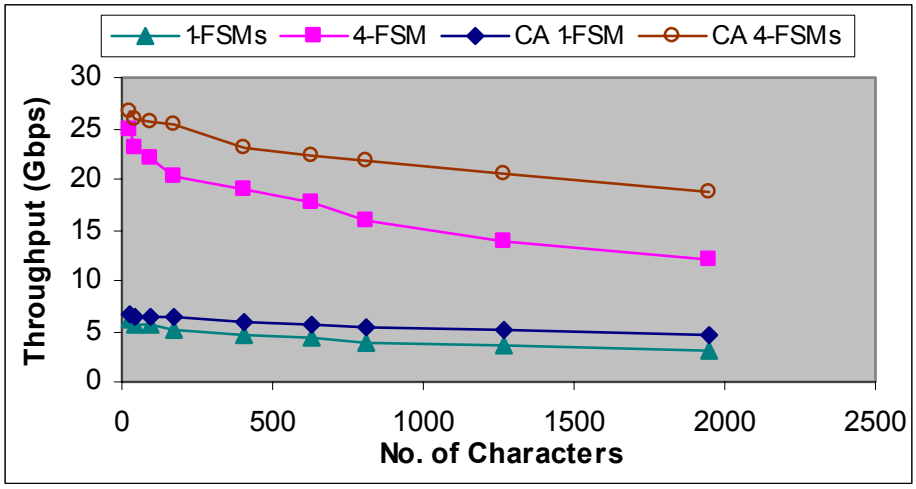


Figure 5.9.Throughput after compression

Figure 5.9 shows the throughput of the accelerator for different number of parallel FSMs both before and after compression. The CACTI tool was set to simulate a 0.13 micron feature size state of the art SRAM. The improvement in performance is quite clear, 4 FSMs will achieve close to 23 Gbps compared to less than 20 Gbps before compression.

5.6 Comparison with Previous Work

Tuck et al. [27] stored the high level nodes including the pointers to the next and failure states in the RAM. Because of that a huge memory of about 53MB was needed to store the Snort rules set. They used the analogy between IP forwarding and string matching to apply

bit-mapping and path compression to the AC tree [52,53], reducing its size to 2.8MB and 1.1MB, respectively. Our approach stores the state tables in the RAM and uses a minimal logic to traverse the tables and find a match. The state tables are around 3MB in size without the use of any compression techniques. We managed to further reduce the database size to 362KB by applying a hardware efficient compression algorithm.

Table 5.3 compares the performance of our design with regular expressions/FAs, discrete comparators and CAMs based designs. The performance numbers for Bloom filters implementation were not available. The throughput numbers for the other designs were obtained from Sourdis *et al.* [4]. The area numbers are estimated based on the number of logic cells used and area of logic cells as reported in FPGA datasheets. The power numbers are calculated from the average logic cell power consumption for the FPGA used (5.9µW/LC per MHz) and where normalized for the same number of characters (806 characters). The numbers for our design are worst case scenario of a large FSM with all the strings in the Snort rules set (22143 characters).

Our design out performed Sidhu's NFAs, Cho's discrete comparators, and Gokhale's CAMs in terms of throughput and power. The area of the RAM based design is quite large that's attributed to the 3 MB needed to accommodate all the rules. The problem is fixed with the compression algorithm and the area of our design beats all the other designs. On the other hand, Sourdis's pre-decoded CAMs used extensive fine grain pipelining to increase the throughput from 2 Gbps to 9.7 Gbps. Our compression algorithms achieves higher

throughput at similar settings. Our design performs at 13.4 Gbps while processing all of the strings in the Snort database. Our design has a lower power because of the use of SRAM as opposed to the use of the power hungry CAMs in Sourdis's design. The area requirements are close and reasonable in both designs.

L. Tan et al. [55], considered a different approach to splitting FSMs, the split the large FSM into eight separate FSMs, each is responsible for one bit on the eight bit input character. The resulting FSMs has only two possible next states as opposed to 256 in the original AC algorithm. The FSMs are loosely coupled and they can be run independently, assuming they can merge the results at the end of execution which proves to be quite challenging. The work is similar to our original design before the application of the compression algorithms and they report similar results. Their design appeared after our work was published [37, 54] and they referenced our papers.

Table 5.3. String matching hardware comparison

Description	Input Bits	Number of Characters	Throughput (Gbps)	Area (mm2)	Power (W)
Aldwairi. Compression algorithm	32	22143	13.4	0.68	0.4
Aldwairi *et al.* State tables in a RAM	32	22143	5.2	5.85	2.6
Sourdis *et al.* Pre-decoded CAMs	32	18032	9.7	0.33	5.1
Gokhale *et al.* CAMs	32	640	2.2	1.4	4.9
Cho *et al.* Discrete Comparators	32	1607	2.9	0.98	2.9
Sidhu *et al.* NFAs/Regular Expression	8	-	0.75	6.1	3

Chapter 6

CONCLUSIONS AND FUTURE WORK

We have studied Snort rules set and shown that 87% of the rules have patterns to match against the packets stream. This further emphasizes the need for hardware acceleration for pattern matching. We introduced a reconfigurable string matching accelerator for intrusion detection. The concept is a memory implementation of the Aho-Corasick FSM where the state tables are directly stored in the RAM rather than the high level tree data structure. We devised an algorithm to classify and split the rules into smaller independent subsets. A separate FSM is generated for every subset, this results in a small memory requirement that is likely to fit in on-chip SRAM. We presented a novel hardware compression algorithm for efficient data storage and fast retrieval. The algorithm reduced the total memory requirement by a factor of 9. The total RAM needed to store the entire Snort rule set is 362KB. We have shown that the accelerator can achieve up to 13Gbps which is sufficient for inline network protection. We also showed that our design out performed the previous work published in the same area.

The resulting contributions of this work are:

1. A pattern matching coprocessor to speed up intrusion detection with a reconfigurable design that can adapt to changing attacks and changing protocols. A tabular RAM based FSM implementation is used boost throughput and maintain configurability.
2. Novel Pattern matching algorithms for splitting large FSMs to several small FSMs that can run in parallel.
3. We present a novel hardware-specific string matching algorithm that can store the Snort rule set in only 0.36 MB. The algorithm is based on compressing all the redundant information in the state tables such as failure pointers to the idle states.
4. The compiler is a novel piece of software that parses the strings, partitions FSMs, creates state tables and compresses the data. The compiler takes only in the order of tens of milliseconds to complete.

The work described in this dissertation has significantly reduced the size of Snort rules set and achieved line rate speeds. Despite the work done, there is still room for improvement. There are many other interesting aspects of intrusion detection that are not addressed:

1. Devise a new classification and load balancing algorithms to split the rule sets into smaller related string sets. We've shown that the smaller the string set the smaller the RAM and the faster the matching algorithm is. Our study of Snort rules suggests

grouping the strings with common prefixes together to maximize the number of common states.

2. Investigate an optimized Aho-Corasick implementation that employs compression to further reduce the memory requirements. The literature on storing Sparse Matrices and Banded Matrices is quite extensive. Compressed Row Storage (CSR) format or Banded-Row Format technology may be used to improve the Aho-Corasick state table storage requirements.
3. Investigate the use of the pattern matching system for deep stateful inspection by considering attacks spread over several packets. The same FSM structure can be used to monitor the connection state and the packet content up to the application layer.
4. One of the most common questions we've been asked when we presented our work to the industry was how to protect against emerging attacks. Signatures based IDSs fall short in that area, because the signatures are created manually after the fact the network has been compromised. Anomaly intrusion detection systems on the other hand, suffer from high rate of false positives and cannot run at wire speed due to the large behavioral database. The opportunity is to research the automatic generation of highly accurate network intrusion detection signatures in real-time or near real-time.
5. Another open problem is to integrate and make sense of all the alerts a security engineer receives on daily basis from the various security systems. A system that

correlates alerts from IDSs and firewalls at different network entry points, as well as system logs and antivirus software.

The intrusion detection area is broad and the door is open for more future work. In a separate project our group is planning to expand the state tables to store predetermined "normal" behaviors to support anomaly-based ID techniques. In addition, the effort includes designing multi-level intrusion detection system using a combination of signature and anomaly detection methods in programmable hardware. The multi-level intrusion detection system uses a hybrid decision making techniques to choose between the signature and anomaly based components to reduce false positives and protect against known and emerging attacks.

BIBLIOGRAPHY

[1] L. Roberts. Internet Growth Trends, in IEEE Computer - Internet Watch, Jan 2000.

[2] R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. In IEEE Symposium on Field Programmable Custom Computing Machines (FCCM01), April 2001.

[3] D. Carver, R. Franklin, and B. Hutchings. Assisting network intrusion detection with reconfigurable hardware. In IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM02), April 2002.

[4] I. Sourdis and D. Pnevmatikatos. Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching. In Proceedings of 12th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM04), April 2004.

[5] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett. Granidt: Towards gigabit rate network intrusion detection technology. In Proceedings of the 12th International Conference on Field-Programmable Logic and Applications, Sept. 2002.

[6] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a network intrusion detection system. In Proceedings of 13th International Conference on Field Programmable Logic and Applications, 2003.

[7] Y. Cho, S. Navab and W. Mangione-Smith. Specialized Hardware for Deep Network Packet Filtering. In Proceedings12th International Conference on Field-Programmable Logic and Applications, Sept. 2002.

[8] D. Dittrich. "Distributed Denial of Service (DDoS) Attacks and Tools". http://staff.washington.edu/dittrich/misc/ddos/.

[9] D. Dittrich. Trinoo Analysis. University of Washington Oct., 1999. http://staff.washington.edu/dittrich/misc/trinoo.analysis.

[10] N. Hastings, P. McLean. TCP/IP Spoofing Fundamentals. In Proceedings of the IEEE Fifteenth Annual International Phoenix Conference on Computer and Communications, 1996, pp 218-224.

[11] Symantec AntiVirus Research Center. http://www.symantec.com/avcenter/index.html

[12] McAfee Virus Information Library. http://vil.mcafee.com/

[13] T. Austin, E. Scott, and S. Gurindar. Efficient detection of all pointer and array access errors. In Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, pages 290–301, June 1994.

[14] G Kayacik, M. Heywood, N. Zincir-Heywood. Evolving Successful Stack Overflow Attacks for Vulnerability Testing. In proceedings of the 21st Annual Computer Security Applications Conference. Dec., 2005.

[15] States Computer Emergency Readiness Team. http://www.us-cert.gov/

[16] S. Axelsson. Intrusion Detection Systems: A Survey and Taxonomy. Chalmers University of Technology, Dept, Sweden, Technical Report 99-15, 2000.

[17] Honeyd Virtual Honeypot hompage, http://www.Honeyd.org.

[18] L. Spitzner, Open Source Honeypots: Learning with Honeyd. http://www.securityfocus.com/infocus/1659.

[19] H. S. Javitz and A. Valdes. The NIDES Statistical Component Description and Justification. Technical report, SRI International, Menlo Park, CA, March 1994.

[20] P. Helman and G. Liepins. Statistical Foundations of Audit Trail Analysis for the Detection of Computer Misuse. In IEEE Transactions on Software Engineering, 19(9), pp. 886–901, 1993.

[21] C. Warrender, S. Forrest, and B.A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In IEEE Symposium on Security and Privacy, pp. 133–145, 1999.

[22] D.Wagner and D. Dean. Intrusion Detection via Static Analysis. In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 2001. IEEE Press.

[23] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In Proceedings of the 1997 IEEE Symposium on Security and Privacy, pp. 175–187, May 1997.

[24] R. Boyer and J. Moore. A fast string searching algorithm. In Communications of the ACM, 20(10), pp762–772, October 1977.

[25] M. Fisk and G. Varghese. Applying Fast String Matching to Intrusion Detection. Sept. 2002.

[26] A. Aho and M. Corasick. Efficient string match-ing: An aid to bibliographic search. In Communications of the ACM, 18(6), pp.333-343, June 1975.

[27] N. Tuck, T. Sherwood, B. Calder and G. Varghese. Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection. In proceedings of the IEEE Infocom conference, March 2004.

[28] Snort.org," http://www.snort.org/.

[29] Shadow homepage. http://www.nswc.navy.mil/ISSEC/CID/.

[30] V. Paxson. Bro: A system for detecting network intruders in real-time. In Computer Networks, 31(23-24), pp. 2435-2463, Dec. 1999.

[31] M. Vision. Advanced reference archive of current heuristics for network intrusion detection systems (arachNIDS), http://www.whitehats.com/ids/.

[32] M. Roesch. Snort - lightweight intrusion detection for networks. In Proceedings of LISA99, the 13th Systems Administration Conference, 1999.

[33] R. Rehman. Intrusion Detection with SNORT: Advanced IDS Techniques Using SNORT, Apache, MySQL, PHP, and ACID. New Jersey: Prentice Hall. May 2003. 75-125.

[34] Snort 2.0 - Detection Revisited. Sourcefire white paper. Feb 2003.

[35] P. Gupta and N. McKeown, “Algorithms for packet classification”, IEEE Network Special Issue, Mar/Apr 2001, vol. 15, no. 2, pp 24-32.

[36] S. Antonatos, K. Anagnostakis, and E. Markatos. Generating realistic workloads for network intrusion detection systems. In ACM Workshop on Software and Performance, 2004.

[37] M. Aldwairi, T. Conte, and P. Franzon. Configurable string matching hardware for speedup up intrusion detection. In Workshop on Architectural Support for Security and Anti-virus (WASSA) Held in Cooperation with ASPLOS XI, Oct. 2004.

[38] J. Moscola, J. Lockwood, R. Loui, M. Pachos. Implementation of a content scanning module for an internet firewall. In Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines. Napa, CA, 2003.

[39] S. Dharmapurikar, M. Attig and J. Lockwood. Design and Implementation of a String Matching System for Network Intrusion Detection using FPGA-based Bloom Filters. In the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '04), April 2004.

[40] S. Li, J. Torresen, O. Sorasen. Exploiting Stateful Inspection of Network Security. In Proceedings of the 13th International Conference on Field Programmable Logic and Application, FPL 2003, Lisbon, pp1153-1157, Portugal, Sept., 2003 .

[41] C. Kruegel, F. Valeur, G. Vigna and R. Kemmerer. Stateful Intrusion Detection for High-Speed Networks, In Proceedings of the 2002 IEEE Symposium on Security and Privacy, Page 285, 2002.

[42] G. Vigna, W. Robertson, V. Kher, and R. Kemmerer. A Stateful Intrusion Detection System for World-Wide Web Servers. In Proceedings of the Annual Computer Security Applications Conference (ACSAC), pp34-43, Las Vegas, NV, Dec. 2003.

[43] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State Transition Analysis: A Rule-Based Intrusion Detection System. In IEEE Transactions on Software Engineering, 21(3), March 1995.

[44] S. Forrest, S.A. Hofmeyr and A. Somayaji. Computer immunology. Communications of the ACM, 40(10), pp 88-96, Oct. 1997.

[45] A. Kosoresow and S. Hofmeyr. Intrusion detection via system call traces. In IEEE Software, 14(5), pp 35-42, Sept. 1997.

[46] A. Ghosh, A. Schwartzbard and M. Schatz. Learning Program Behavior Profiles for Intrusion Detection. In Proceedings of Workshop on Intrusion Detection and Network Monitoring. pp. 51-62. Santa Clara, California, April, 1999.

[47] C. Michael, Anup K. Ghosh: Simple, state-based approaches to program-based anomaly detection. ACM Transaction on Information System Security, 5(3), pp. 203-237, 2002.

[48] OpenCores.org OpenRISC 1000 pages. http://www.opencores.org/pnews.cgi/list/or1k?no_loop=yes.

[49] B. Iyer, T. Conte. Clustered Length Architecture Word processor (CLAW). Opencores.org. Sept. 2004. http://www.opencores.org/cvsweb.shtml/claw/.

[50] CACTI http://www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html.

[51] CACTI 4.2 Web Interface. http://quid.hpl.hp.com:9081/cacti/sram.y

[52] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In Proceedings of SIGCOMM, pages 3–14, 1997.

[53] W. Eatherton. Hardware-based internet protocol prefix lookups. Washington University Electrical Engineering Department (MS Thesis), May 1999.

[54] M. Aldwairi, T. Conte, and P. Franzon. Configurable String Matching Hardware for Speeding up Intrusion Detection. ACM SIGARCH Computer Architecture News, 33(1):99–107, 2005.

[55] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05). pp. 112-122, 2005

APPENDICES

8.1 Appendix A ac.cpp

```
/*****************************************************************
*
* NAME: ac.cpp
*
* DESCRIPTION:
*                 This code reads a set of strings, builds up an AC tree,
*     traverses the tree and generates a state table.
*
*
*
* COMPILE:
* -----------
*     The simulator can be complied in several modes
*
*           <eos> make clean
*           <eos> make mode=[MODE]
*
* MODE:
*
*     Depending on the mode of compilation the code runs differently:
*
*     1. In NORMAL mode the program will read the parameters from the
*         command line, parse the strings and store the SRAM in mem.
*     2. In DEBUG mode, it does everything NORMAL mode and it prints
*         Out debugging information (AC tree and SRAM content)
*     3. In PACKET mode, it does everything as in NORMAL mode and reads
*           a packets from a file and matches it against the string set.
*           it prints a trace showing matches...
*     4. In TIME mode, it does everything as in NORMAL mode and measures
*           the time to generate the state tables
*     5. In VERILOG mode, it does everything as in NORMAL mode and reads
*           a packets from a file and matches it against the string set.
*           it prints a verilog sequence to test match.v
*
* Run:
* -------
*     1. To run in PACKET or VERILOG mode
*           <eos> ac [file1] [file2]
*                       file1: Strings file name
*                       file2: Packet payload file name
*     2. To run in all other modes
*           <eos> ac [file1]
*                       file1: Strings file name
*
*     3. To get help regardless of mode,type
```

```
*           <eos> ac
*
*
* REVISION HISTORY
*     Date        Programmer              Description
*     4/04  - Created the basic AC implementation (Monther Aldwairi)
*     6/04  - Modified the memory allocations (Monther Aldwairi)
*     7/04  - Added states, tree traversal, printing and args functons
*   8/04    - Added FSM tables generation modules, and RAM simulator
*     12/04 - Added functions to process packets, hex2dec  converters
*                       and parse strings (Monther Aldwairi)
*     4/05  - Added time measurement directives (Monther Aldwairi)
*     6/05  - Wrapped it and added running modes (Monther Aldwairi)
*
\*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include "ac.h"


void traverse(AC_TREE root)
{
      printState(root);

      if (root->children !=NULL)
            traverse(root->children);
      if (root->sibling != NULL)
            traverse(root->sibling);
}

void printNode(AC_TREE root)
{
      if (root->matchid != 0)
            printf("%c\n", root->ch);
      else
            printf("%c", root->ch);
}

//Recursion over all siblings of a node

void printSibling(AC_TREE node)
{
      printf("\t%c:%s",node->ch, node->state);
      if (node->sibling!= NULL)
            printSibling(node->sibling);
}
```

```
//prints a node/state and all state transitions

void printState(AC_TREE root)
{
      printf("%s",root->state);
      printf(":%x",root->binaryState);

      if (root->children !=NULL)
      {
            printf("\t\t%c:%s",root->children->ch, root->children->state);
            if (root->children->sibling != NULL)
                  printSibling(root->children->sibling);
            else
                  printf("\t\t");
      }
      else
            printf("\t\t\t\t");

      if (root->faillink != NULL)
            printf("\tff:%s", root->faillink->state);
      printf("\n");
}

/*
*     traverse the tree filling the FSM table
*
*/
void populate(AC_TREE root)
{
      populateRAM(root);

      if (root->children !=NULL)
            populate(root->children);
      if (root->sibling != NULL)
            populate(root->sibling);
}

//Traverse the tree and fill the RAM model with the state table

void populateRAM(AC_TREE root)
{
      id[root->binaryState] = root->matchid;
      if (root->children !=NULL)
      {
            FSM[root->binaryState][(unsigned int)(root->children->ch)] =
root->children->binaryState;
            if (root->children->sibling != NULL)
                  fillSibling(root->children->sibling, root->binaryState);

      }
}
```

```
void fillSibling(AC_TREE node, int state)
{
      FSM[state][(unsigned int)(node->ch)] = node->binaryState;
      if (node->sibling!= NULL)
            fillSibling(node->sibling,state);
}

//add failure states to the RAM

void failure(AC_TREE root)
{
      addFail(root);

      if (root->children !=NULL)
            failure(root->children);
      if (root->sibling != NULL)
            failure(root->sibling);
}

//Traverse the tree and fill the RAM model with the state table

void addFail(AC_TREE root)
{
      int j=0;

      if (root->faillink == NULL)
      {//copy idle state links
            for (j=0; j<256; j++)
                  if (FSM[root->binaryState][j] == 0)
                        if (FSM[0][j] != 0)
                        {
                              FSM[root->binaryState][j] = FSM[0][j];
                        singleCharState++;
                        }
      }
      else//follow the link and then traverse tree for all chars
      {
            for (j=0; j<256; j++)
            {
                  if (FSM[root->binaryState][j] == 0)
                        FSM[root->binaryState][j] = FSM[root->faillink-
>binaryState][j];
                  //if still zero then copy idle states
                  if (FSM[root->binaryState][j] == 0)
                        if (FSM[0][j] != 0)
                        {
                              FSM[root->binaryState][j] = FSM[0][j];
                  singleCharState++;
                        }
            }
```

```
      }
}

//process the command line parameters

void parseArgs(int argc, const char **argv)
{

#ifdef PACKET
      if (argc != 3)
      {
            printf("usage: %s [file1] [file2]", argv[0]);
            printf("\n\tfile1: Strings file name");
            printf("\n\tfile2: Packet payload file name\n");
        exit(1);
    }

      if ( (argv[2] == NULL) || ((fpackets=fopen(argv[2],"r")) == NULL))
    {
        printf("\n %s: file doesn't exist\n", argv[2]);
        exit(1);
    }

#endif

#ifdef VERILOG

      if (argc != 3)
      {
            printf("usage: %s [file1] [file2]", argv[0]);
            printf("\n\tfile1: Strings file name");
            printf("\n\tfile2: Packet payload file name\n");
        exit(1);
    }

      if ( (argv[2] == NULL) || ((fpackets=fopen(argv[2],"r")) == NULL))
    {
        printf("\n %s: file doesn't exist\n", argv[2]);
        exit(1);
    }

      if ( (fverilog=fopen("testIDS.v","w+")) == NULL)
      {
            printf("\n cannot creat testIDS.v\n");
        exit(1);
      }

#endif

#ifndef PACKET
```

```
#ifndef VERILOG


      if (argc != 2)
      {
            printf("usage: %s [file1]", argv[0]);
            printf("\n\tfile1: Strings file name\n");
        exit(1);
    }

#endif
#endif

      if ( (argv[1] == NULL) || ((fstrings=fopen(argv[1],"r")) == NULL))
    {
        printf("\n %s: file doesn't exist\n", argv[1]);
        exit(1);
    }

}

//process Snort rule content for spaces and Hex string and
// converts them into ASCII string

void parseString(char *string, char *newStr)
{
      int i,j,begin,end,index;
      char hex[500];
      i=j=begin=end=index=0;

      for(i=0; i<strlen(string) - 1; i++)
      {
                  if (string[i] == '|')
                  {
                        begin = (begin==0) ? 1 : 0;
                        end     = (begin==0) ? 1 : 0;
                  }
                  if (end == 1)
                  {
                        begin = end = 0;
                        newStr = strcat(newStr,createAsciiFromHex(hex));
                        j = strlen(newStr);
                        index = 0;
                  }
                  else if (begin == 0)
                  {
                        newStr[j++]= string[i];
                        newStr[j]= '\0';
                  }
                  else
                  {
```

```
                        if (string[i] != ' ' && string[i] != '|')
                        {
                              hex[index++] = string[i];
                              hex[index]='\0';
                        }
                  }
      }

}



/*
 * ac_allocate*
 * Creates and initializes a new AC_STRUCT structure
 *
 */
AC_STRUCT *(void)
{
  AC_STRUCT *node;
  if ((node = (AC_STRUCT *) (malloc(sizeof(AC_STRUCT)))) == NULL)
    return NULL;
  memset(node, 0, sizeof(AC_STRUCT));

  if ((node->tree = (ACTREE_NODE *) (malloc(sizeof(ACTREE_NODE)))) == NULL)
{
    free(node);
    return NULL;
  }
  memset(node->tree, 0, sizeof(ACTREE_NODE));

  return node;
}


/*
 * ac_add_string
 *
 * Adds a string to the AC_STRUCT keyword tree.
 *
 * Parameters:   node      -  an AC_STRUCT structure
 *               P         -  the sequence
 *               M         -  the sequence length
 *               id        -  the sequence identifier
 *
 * Returns:  non-zero on success, zero on error.
 */

int ac_add_string(AC_STRUCT *node, char *P, int M, int id)
{
  int i, j, k, newsize;
```

```
  AC_TREE tnode, child, back, newnode, list, tail;

  /*
   * Return a zero if a previous error had occurred, or if the
   * given id equals zero.  An id value of zero is used by the
   * algorithm to signal that no pattern ends at a node in the
   * keyword tree.  So, it can't be used as a pattern's id.
   */
  if (node->errorflag || id == 0)
    return 0;

  P--;

  /*
   * Allocate space for the new string's information.
   */
  if (node->Psize <= id) {
    if (node->Psize == 0) {
      newsize = (id >= 16 ? id + 1 : 16);
      node->Plengths = (int *) malloc(newsize * sizeof(int));

    }
    else {
      newsize = node->Psize + id + 1;
      node->Plengths = (int *) realloc(node->Plengths, newsize *
sizeof(int));
    }
    if (node->Plengths == NULL) {
      node->errorflag = 1;
      return 0;
    }

    for (i=node->Psize; i < newsize; i++)
      node->Plengths[i] = 0;
    node->Psize = newsize;
  }

  if (node->Plengths[id] != 0) {
    fprintf(stderr, "Error in Aho-Corasick preprocessing.  "
            "Duplicate identifiers\n");
    return 0;
  }

  /*
   * Add the string to the keyword tree.
   */
  tnode = node->tree;
  for (i=1; i <= M; i++) {
    /*
     * Find the child whose character is P[i].
     */
```

```
    back = NULL;
    child = tnode->children;
    while (child != NULL && child->ch < P[i]) {
      back = child;
      child = child->sibling;
    }

    if (child == NULL || child->ch != P[i])
      break;

    tnode = child;

#ifdef STATS
    node->prep_old_edges++;
#endif
  }

  /*
   * If only part of the pattern exists in the tree, add the
   * rest of the pattern to the tree.
   */
  if (i <= M) {
    list = tail = NULL;
    for (j=i; j <= M; j++) {
      if ((newnode = (ACTREE_NODE *) malloc(sizeof(ACTREE_NODE))) == NULL)
        break;
      memset(newnode, 0, sizeof(ACTREE_NODE));
      newnode->ch = P[j];

        for (k=0; k<j; k++)
          newnode->state[k] = P[k+1];
       //binary state assignment/encoding!
        newnode->binaryState= states++;

      if (list == NULL)
        list = tail = newnode;
      else
        tail = tail->children = newnode;

#ifdef STATS
      node->prep_new_edges++;
#endif
    }
    if (j <= M) {
      while (list != NULL) {
        tail = list->children;
        free(list);
        list = tail;
      }
      return 0;
    }
```

```
    list->sibling = child;
    if (back == NULL)
      tnode->children = list;
    else
      back->sibling = list;

    tnode = tail;
  }

  tnode->matchid = id;
  node->Plengths[id] = M;
  node->ispreprocessed = 0;

  return 1;
}


/*
 * ac_del_string
 *
 * Deletes a string from the keyword tree.
 *
 * Parameters:   node  -  an AC_STRUCT structure
 *               P     -  the sequence to be deleted
 *               M     -  its length
 *               id    -  its identifier
 *
 * Returns:  non-zero on success, zero on error.
 */

int ac_del_string(AC_STRUCT *node, char *P, int M, int id)
{
  int i, flag;
  AC_TREE tnode, tlast, tback, child, back;

  if (node->errorflag || id > node->Psize || node->Plengths[id] == 0)
    return 0;

  P--;            /* Shift to make sequence be P[1],...,P[M] */

  /*
   * Scan the tree for the path corresponding to the keyword to be deleted.

   */
  flag = 1;
  tlast = tnode = node->tree;
  tback = NULL;

  for (i=1; i <= M; i++) {
    /*
```

```
   * Find the child matching P[i].  It must be there.
   */
  child = tnode->children;
  back = NULL;
  while (child != NULL && child->ch != P[i]) {
    back = child;
    child = child->sibling;
  }

  if (child == NULL) {
    fprintf(stderr, "Error in Aho-Corasick preprocessing.  String to be "

            "deleted is not in tree.\n");
    return 0;
  }


  if (i < M && child->matchid != 0) {
    flag = 1;
    tlast = child;
  }
  else if (back != NULL || child->sibling != NULL) {
    flag = 2;
    tlast = child;
    tback = (back == NULL ? tnode : back);
  }

  tnode = child;
}

if (tnode->children != NULL) {
  tnode->matchid = 0;
}
else {
  if (flag == 1) {
    child = tlast->children;
    tlast->children = NULL;
    tlast = child;
  }
  else {
    if (tback->children == tlast)
      tback->children = tlast->sibling;
    else
      tback->sibling = tlast->sibling;
  }

  while (tlast != NULL) {
    child = tlast->children;
    free(tlast);
    tlast = child;
  }
```

```
  }

  node->Plengths[id] = 0;
  node->ispreprocessed = 0;

  return 1;
}


/*
 * ac_prep
 *
 * Compute the failure and output links for the keyword tree.
 *
 * Parameters:  node  -  an AC_STRUCT structure
 *
 * Returns: non-zero on success, zero on error.
 */
int ac_prep(AC_STRUCT *node)
{
  char x;
  AC_TREE v, vprime, w, wprime, root, front, back, child;

  if (node->errorflag)
    return 0;


  root = node->tree;

  front = back = root;
  front->faillink = NULL;
  front->outlink = NULL;

  while (front != NULL) {
    v = front;
    x = v->ch;
    vprime = v->outlink;

    /*
     * Add the node's children to the queue.
     */
    for (child=v->children; child != NULL; child=child->sibling) {
      child->outlink = v;
      back->faillink = child;
      back = child;
    }
    back->faillink = NULL;

    front = front->faillink;
    v->faillink = v->outlink = NULL;
```

```
    /*
     * Set the failure and output links.
     */
    if (v == root)
      ;
    else if (vprime == root)
      v->faillink = root;
    else {
      /*
       * Find the find link in the failure link chain which has a child
       * labeled with x.
       */
      wprime = NULL;
      w = vprime->faillink;

      while (1) {
        wprime = w->children;
        while (wprime != NULL && wprime->ch < x)
          wprime = wprime->sibling;

        if ((wprime != NULL && wprime->ch == x) || w == root)
          break;

        w = w->faillink;

#ifdef STATS
        node->prep_fail_compares++;
#endif
      }
#ifdef STATS
      node->prep_fail_compares++;
#endif

      if (wprime != NULL && wprime->ch == x)
        v->faillink = wprime;
      else
        v->faillink = root;

      if (v->matchid != 0) {
        if (v->faillink->matchid != 0)
          v->outlink = v->faillink;
        else
          v->outlink = v->faillink->outlink;
      }
    }
  }

  node->ispreprocessed = 1;
  node->initflag = 0;

  return 1;
```

```
}

/*
 * ac_search_init
 *
 * Initializes the variables used during an Aho-Corasick search.
 * See ac_search for an example of how it should be used.
 *
 * Parameters:  node  -  an AC_STRUCT structure
 *              T     -  the sequence to be searched
 *              N     -  the length of the sequence
 */

void ac_search_init(AC_STRUCT *node, char *T, int N)
{
  if (node->errorflag)
    return;
  else if (!node->ispreprocessed) {
    fprintf(stderr, "Error in Aho-Corasick search.  The preprocessing "
            "has not been completed.\n");
    return;
  }

  node->T = T - 1;          /* Shift to make sequence be T[1],...,T[N] */
  node->N = N;
  node->c = 1;
  node->w = node->tree;
  node->output = NULL;
  node->initflag = 1;
  node->endflag = 0;
}


/*
 * ac_search
 *
 * Scans a text to look for the next occurrence of one of the patterns
 * in the text.
 *
 * Parameters:  node           -  a preprocessed AC_STRUCT structure
 *              length_out     -  where to store the new match's length
 *              id_out         -  where to store the identifier of the
 *                                pattern that matched
 *
 * Returns:  the left end of the text that matches a pattern, or NULL
 *           if no match occurs.
 */
char *ac_search(AC_STRUCT *node, int *length_out, int *id_out)
{
  int c, N, id;
  char *T;
```

```
  AC_TREE w, wprime, root;

  if (node->errorflag)
    return NULL;
  else if (!node->ispreprocessed) {
    fprintf(stderr, "Error in Aho-Corasick search.  The preprocessing "
            "has not been completed.\n");
    return NULL;
  }
  else if (!node->initflag) {
    fprintf(stderr, "Error in Aho-Corasick search.  ac_search_init was not
"
            "called.\n");
    return NULL;
  }
  else if (node->endflag)
    return NULL;

  T = node->T;
  N = node->N;
  c = node->c;
  w = node->w;
  root = node->tree;

  /*
   * If the last call to ac_search returned a match, check for another
   * match ending at the same right endpoint (denoted by a non-NULL
   * output link).
   */
  if (node->output != NULL) {
    node->output = node->output->outlink;
#ifdef STATS
    node->outlinks_traversed++;
#endif

    if (node->output != NULL) {
      id = node->output->matchid;
      if (id_out)
        *id_out = id;
      if (length_out)
        *length_out = node->Plengths[id];

      return &T[c] - node->Plengths[id];
    }
  }

  /*
   * Run the search algorithm, stopping at the first position where a
   * match to one of the patterns occurs.
   */
  while (c <= N) {
```

```
    /*
     * Try to match the next input character to a child in the tree.
     */
    wprime = w->children;
    while (wprime != NULL && wprime->ch != T[c])
      wprime = wprime->sibling;

#ifdef STATS
    node->num_compares++;
#endif

    /*
     * If the match fails, then either use the failure link (if not
     * at the root), or move to the next character since no prefix
     * of any pattern ends with character T[c].
     */
    if (wprime == NULL) {
      if (w == root)
        c++;
      else {
        w = w->faillink;

#ifdef STATS
        node->num_failures++;
#endif
      }
    }
    else {
      /*
       * If we could match the input, move down the tree and to the
       * next input character, and see if that match completes the
       * match to a pattern (when matchid != 0 or outlink != NULL).
       */
      c++;
      w = wprime;

#ifdef STATS
      node->edges_traversed++;
#endif

      if (w->matchid != 0)
        node->output = w;
      else if (w->outlink != NULL) {
        node->output = w->outlink;

#ifdef STATS
        node->outlinks_traversed++;
#endif
      }

      if (node->output != NULL) {
```

```
        id = node->output->matchid;
        if (id_out)
          *id_out = id;
        if (length_out)
          *length_out = node->Plengths[id];

        node->w = w;
        node->c = c;

        return &T[c] - node->Plengths[id];
      }
    }
  }

  node->c = c;
  node->endflag = 1;

  return NULL;
}


/*
 * ac_free
 *
 * Free up the allocated AC_STRUCT structure.
 *
 * Parameters:   node  -  a AC_STRUCT structure
 *
 * Returns:  nothing.
 */
void ac_free(AC_STRUCT *node)
{
  AC_TREE front, back, next;

  if (node == NULL)
    return;

  if (node->tree != NULL) {
    front = back = node->tree;
    while (front != NULL) {
      back->sibling = front->children;
      while (back->sibling != NULL)
        back = back->sibling;

      next = front->sibling;
      free(front);
      front = next;
    }
  }

  if (node->Plengths != NULL)
```

```
    free(node->Plengths);

  free(node);
}


/*
 * main
 *
 * Generates AC tree, search AC tree, generate state tables,
 * measure execution time, measure string length and number
 *of states....
 *
 */

main(int argc, const char **argv)
{
      AC_STRUCT *node;
      int length_out = 0;
      int id_out = 0;

      char ch, string[500],newString[500];
      int j, i=0;
      unsigned int matchID, state = 0;

#ifdef TIME
      struct timeval tv1, tv2;
      tv1.tv_usec=tv2.tv_usec=0;
#endif

      parseArgs(argc,argv);



      //initialize FSM
      for (i=0; i< 10000; i++)
      {
            id[i] = 0;
            for (j=0; j<256; j++)
                  FSM[i][j] = 0;
      }

      //allocate memory for the tree
      if ((node = ac_allocate()) == NULL) {
            free(node);
            exit(0);
      }

    i=1;
      int avg_string_length = 0;
      int num_strings=0;
```

```
	stateTransCount = 0;
	singleCharState = 0;


#ifdef TIME
	gettimeofday(&tv1, NULL);
#endif

	while(fgets(string, 1024, fstrings)!= NULL)
	{
		parseString(string, newString);
		ac_add_string(node, newString, strlen(newString), i++);
		avg_string_length += strlen(newString);
		num_strings++;
		newString[0]='\0';
	}

	ac_prep(node);
	//ac_search_init(node, string, length);
	populate(node->tree);
	failure(node->tree);
	failure(node->tree);

#ifdef DEBUG
	traverse(node->tree);
#endif


#ifdef TIME
	//time estimation
	gettimeofday(&tv2, NULL);
	printf("\nTime Elapsed %d secs\n",(int)(tv2.tv_usec-tv1.tv_usec));
#endif


#ifdef PACKET

// receives the packet payload one character at a time and access the RAM
state table
	state = 0;

	while(fscanf(fpackets,"%c", &ch) != EOF)
	{
		if (ch != 0x0D && ch != 0x0A)
		{
			printf("\n%x->%c",state, ch);
			//if (id[state] > 0)
				printf("\tid:%d",id[state]);
			state= FSM[state][(unsigned int) ch];
		}
	}
```

```
#endif

#ifdef DEBUG

      //prints out the TREE state table

      for (i=0; i< 20; i++)
      {
            printf("\n%d:%x:%x", i, i , id[i]);
            for (j=32; j<256; j++)
                  if (FSM[i][j] != 0)
                        printf(" %c:%x",j, FSM[i][j]);
      }

      //Count state transitions
      for (i=0; i< states; i++)
            for (j=32; j<256; j++)
                  if (FSM[i][j] != 0)
                        stateTransCount++;

      printf("\n");
      printf("\n # string = %8d",num_strings);
      printf("\nAvg length= %8.1f\n",(double)
avg_string_length/num_strings);
      printf("\n%d\t%.1f\t%d\n",num_strings, (double)
avg_string_length/num_strings, states-1);
      printf("\nallState-idleStates=%d-%d=%d\n",stateTransCount,
singleCharState, stateTransCount-singleCharState);

#endif

#ifdef VERILOG
// receives the packet payload one character at a time, access the RAM
state table
// and creates a verilog test file

      printVerilog(1," ",' ',0);
      state = 0;

      while(fscanf(fpackets,"%c", &ch) != EOF)
      {
            if (ch != 0x0D && ch != 0x0A)
            {
                  printVerilog(3,"\n\t\t#10 packet_content\t\t= 8'h",ch,
0);
                  printVerilog(5,";",' ',0);
                  printVerilog(4,"\n\t\t    tree_data\t\t\t= 31'h",' ',
id[state]);
                  printVerilog(4,"0000",' ', state);
                  printVerilog(5,";",' ',0);
```

```
                state= FSM[state][(unsigned int) ch];
            }
        }

        printVerilog(2," ",' ',0);

#endif



    fclose(fstrings);
#ifdef PACKET
    fclose(fpackets);
#endif
#ifdef VERILOG
    fclose(fpackets);
    fclose(fverilog);
#endif
}



//converts hex string to ASCII string

char *createAsciiFromHex(char *hex)
{
        // Every 2 hex characters correspond to an ascii character

        char *digits = "0123456789abcdef";
        int length = strlen(hex);
        int ldigit, rdigit, temp;
        int i,j;

        // The first step will be to verify the hex string is a multiple of 2

        if ( length % 2 ) {
                fprintf(stderr, "createAsciiFromHex: hex string not mod 2");

                return NULL;
        }

        // Now create a new buffer to hold the ascii version
        char *ascii = new char[(length / 2) + 1];
        ascii[length/2] = 0;

        for (i=0; i < (length/2); i++) {
                // clear this entry of the ascii buffer
                ascii[i] &= 0x00;
                ldigit = 0;
                rdigit = 0;
```

```
            // Determine what value the left digit is
            for (j=0; j<16; j++) {
                  // See if the char matches an element of digits
                  if ( hex[i*2] == digits[j] ) {
                        // We found the character so set it's value to the
index
                        //    of digits, or just j
                        ldigit = j;
                  }
            }

            // Determine what value the right digit is
            for (j=0; j<16; j++) {
                  // See if the char matches an element of digits
                  if ( hex[i*2+1] == digits[j] ) {
                        // We found the character so set it's value to the
index
                        //    of digits, or just j
                        rdigit = j;
                  }
            }

            // now temp should be 16*ldigit + lright
            temp = (ldigit * 16) + rdigit;

            // Now we have the decimal value of the original ascii
character
            //    so set this bit pattern in the ascii string
            ascii[i] |= temp;

      }

      return (ascii);
}


void printVerilog(int mode, char *text, char ch, int state_ID)
{

      if (mode == 1)
      {
            fprintf(fverilog, "`include \"match.v\"\n\n");
            fprintf(fverilog, "module test;\n\n");
            fprintf(fverilog, "\treg\t\tclk, reset;\n");
            fprintf(fverilog, "\treg\t\t[7:0]\tpacket_content;\n");
            fprintf(fverilog, "\treg\t\t[31:0]\ttree_data;\n");
            fprintf(fverilog, "\twire\t\t[8:0]\tmatch_id;\n");
            fprintf(fverilog, "\twire\t\t[31:0]\tReadAddress;\n\n");
            fprintf(fverilog, "\tinitial\n");
            fprintf(fverilog, "\tbegin\n\n");
```

```
            fprintf(fverilog, "\t\t$shm_open(\"waves.shm\");\n");
            fprintf(fverilog, "\t\t$shm_probe(\"AS\");\n\n");
            fprintf(fverilog, "\t\treset\t\t\t\t= 1'b0;\n");
            fprintf(fverilog, "\t\tclk\t\t\t\t= 1'b1;\n");
            fprintf(fverilog, "\t\t#5 reset\t\t\t= 1'b1; \n");
            fprintf(fverilog, "\t\t#10 reset\t\t\t= 1'b0;\n\n");
            fprintf(fverilog, "\t\t// a Sample RAM/FSM \n\n");
      }
      else if ( mode == 2)
      {
            fprintf(fverilog, "\n\n\t\t$shm_close();\n");
            fprintf(fverilog, "\t\t$finish;\n");
            fprintf(fverilog, "\t\tend\n");
            fprintf(fverilog, "\talways #5 clk = ~clk;\n");
            fprintf(fverilog, "\tmatch matching(\n");
            fprintf(fverilog, "\t\t.clk(clk),\n");
            fprintf(fverilog, "\t\t.reset(reset),\n");
            fprintf(fverilog, "\t\t.packet_content(packet_content),\n");
            fprintf(fverilog, "\t\t.tree_data(tree_data),\n");
            fprintf(fverilog, "\t\t.match_id(match_id),\n");
            fprintf(fverilog, "\t\t.ReadAddress(ReadAddress)\n");
            fprintf(fverilog, "\t);\n");
            fprintf(fverilog, "endmodule\n");

      }
      else if (mode == 3)
      {
            fprintf(fverilog, text);
            fprintf(fverilog, "%02x",ch);
      }
      else if (mode ==4)
      {
            fprintf(fverilog, text);
            fprintf(fverilog, "%02x",state_ID);
      }
      else
            fprintf(fverilog, text);
}
```

8.2 Appendix B ac.h

```
#ifndef _AC_H_
#define _AC_H_

void populate(AC_TREE root);
void traverse(AC_TREE root);
void printNode(AC_TREE root);
void printSibling(AC_TREE node);
void printState(AC_TREE root);
void populateRAM(AC_TREE root);
void fillSibling(AC_TREE node, int state);
AC_STRUCT *ac_allocate(void);
int ac_add_string(AC_STRUCT *node, char *P, int M, int id);
int ac_del_string(AC_STRUCT *node, char *P, int M, int id);
int ac_prep(AC_STRUCT *node);
void ac_search_init(AC_STRUCT *node, char *T, int N);
char *ac_search(AC_STRUCT *node, int *length_out, int *id_out);
void ac_free(AC_STRUCT *node);
void failure(AC_TREE root);
void addFail(AC_TREE root);
void parseString(char * string, char *newStr);
char *createAsciiFromHex(char *hex);
void parseArgs(int argc, const char **argv);
void printVerilog(int mode, char *text, char ch, int state_ID);



typedef struct actreenode {
  char ch;
  char state[500];
  unsigned int binaryState;
  int matchid;
  struct actreenode *outlink, *faillink;
  struct actreenode *children, *sibling;
} ACTREE_NODE, *AC_TREE;


typedef struct {
  AC_TREE tree;
  int ispreprocessed, errorflag;

  int Psize;
  int *Plengths;
```

```
  char *T;
  int N, c, initflag, endflag;
  AC_TREE w, output;

  int prep_new_edges, prep_old_edges, prep_fail_compares;
  int num_compares, num_failures, edges_traversed, outlinks_traversed;
} AC_STRUCT;



//Global variables
unsigned int states = 1;
unsigned int FSM[10000][256];
unsigned int id[10000];
unsigned int stateTransCount = 0;
unsigned int singleCharState = 0;

FILE *fstrings;
FILE *fpackets;
FILE *fverilog;
#endif
```