High Speed Deep Packet Inspection with Hardware Support

Fang Yu

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-156
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-156.html

November 22, 2006

High Speed Deep Packet Inspection with Hardware Support

by

Fang Yu

B.S. (Fudan University) 2000
M.S. (University of California, Berkeley) 2002

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Randy H. Katz, Chair
Professor David A. Patterson
Professor Dorit S. Hochbaum

Fall 2006

The dissertation of Fang Yu is approved:

_____

Professor Randy H. Katz, Chair Date

_____

Professor David A. Patterson Date

_____

Professor Dorit S. Hochbaum Date

University of California, Berkeley

Fall 2006

High Speed Deep Packet Inspection with Hardware Support

by

Fang Yu

ABSTRACT

High Speed Deep Packet Inspection with Hardware Support

by

Fang Yu

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Randy H. Katz, Chair

In this dissertation, we developed high speed packet processing algorithms for new services such as network intrusion detection, high speed firewalls, Network Address Translation (NAT), Hypertext Transfer Protocol (HTTP) load balancing, Extensible Markup Language (XML) processing, and Transmission Control Protocol (TCP) offloading. These new services have stringent requirements for speed, extensibility, scalability, and cost-effectiveness. For example, some services require rapid scanning of packets against thousands of known patterns. Traditional packet handling techniques, such as next hop forwarding, focus on packet headers only and fail to support these demanding requirements. This thesis research aims to provide fast and efficient *deep packet inspection techniques* that can function on the entire packet content rather than just the header. To keep up with high speed packet processing in existing networks, we proposed deep packet inspection schemes that are optimized for new technologies such as Ternary Content Addressable Memory (TCAM) and multi-core processors. We propose algorithms that work both on packet headers and packet payload. Our techniques form a cohesive architecture that can perform Gigbit rate packet scanning against thousands of sophisticated patterns.

_____

Professor Randy H. Katz, Chair Date

ACKNOWLEDGEMENTS

I want to sincerely thank my research advisor Professor Randy Katz for taking me as his student and supporting me during the past five years. Although he had more than ten students, he put tremendous efforts on each of us. He met me every week individually, providing detailed guidance on research direction, methodology and making suggestions on academia career. When I need his advice or help, he is always there. I feel even more indebted to the countless hours that he spent during the weekends and holidays improving my writing and presentations. Without his invaluable support, I could not have completed this dissertation.

Besides my advisor, I would like to thank the rest of my qualifying exam committee: Professors Dorit S. Hochbaum, David A. Patterson, and Ion Stoica for asking me constructive questions, giving me insightful comments and reviewing the dissertation.

I want to thank Dr. T.V. Lakshman for introducing me into the field of deep packet inspection. Without his encouragement and guidance, I wouldn't be able to finish the Ph.D. program so quickly. My gratitude also goes to my co-authors: Yanlei Diao for inspiring conversations on research and invaluable advices on my future academic career path; Zhifeng Chen for his support and hard work that led to the development of our regular expression scanning system; Martin Austin Motoyama for his help in writing and conducting tests on our SSA system.

Special thanks go to Professor Aoying Zhou, Hongjun Lu, and Dr. Wenwu Zhu, who introduced me to the system research field and encouraged me to pursue my Ph.D. study at UCB in the first place. I am also very grateful to my mentors in AT&T Rakesh Sinha, John Strand, Bob Doverspike and Angela Chiu for their guide and support during the early years of my Ph.D. study.

I would like to express my sincere gratitude to my groupmate Matthew Caeser for reading most of my papers and thesis and giving me insightful suggestions. My appreciation also goes to other members of the networking group at UC Berkeley. I acknowledge Sharad Agarwal, Matthew Caesar, Yan Chen, Weidong Cui, Ling Huang, Karthik Lakshminarayanan, Boon Thau Loo, Jayanthkumar Kannan, Sridhar Machiraju, George Porter, Ananth Rao, Mukund Seshadri, Lakshminarayanan Subramanian, Shelley Zhuang for their valuable insight, patience for numerous questions, and entertainment after work.

Special thanks to Li Yin, for being a wonderful groupmate and friend, for listening to my complaints and frustrations during the past five years. I would like to thank the following friends for making my years of study at UC Berkeley an enjoyable and memorable journey: Yaping Li, Ning Ma, Yanmei Li, Minghua Chen, Wei Wei, Jianhui Zhang, Rui Xu, Dan Huang, Li Zhuang, Shuheng Zhou, and Jingyang Li. I am also grateful for my friends Ming Chen, Songting Chen, Chong Luo, Ye Fan, Yong Gao, Liping Ke, Lin Qiao, Li Rui, Erming Sun, Li Wei, Qin Xin, Na Yang, Qiu Zhang, Guangyu Zhu for making the life studying aboard a fun experience.

The research work was made possible by the financial support from many agencies and groups including the National Science Foundation, AT&T, INTT MCL, HP, Cisco, Microsoft, and the UC MICRO program.

Last but not least, I am deeply indebted to my parents and sister for their unconditional love, everlasting faith in me, and encouraging support for my years of pursuit in academic career. I would also like to thank my husband Geozheng Ge for his love. His support and encouragement was in the end what made this dissertation possible. My family gave me the strength to overcome the innumerable obstacles that I encountered during my PhD study. I dedicate this dissertation to them.

To my family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

1.1 Motivation

Today's Internet is by no means a safe place: hackers sniff network traffic and expend considerable resources on attacking user's computers, fraudulent phishing activities are on the rise, and worms and viruses cause service disruptions with enormous economic impact. Counter-intuitively, recent increases in network bandwidth and computing performance make things even worse—such increases allow malicious attacks to spread even faster. The recently introduced Slammer worm is one of the fastest spreading computer worms in history [1], infecting more than 75,000 hosts within 10 minutes. Slammer exploits two "buffer overflow" security holes in Microsoft SQL Server 2000 to infect vulnerable hosts. The infected hosts launch Denial of Service (DoS) attacks that dramatically slow normal Internet traffic. It caused around 1 billion USD of damage worldwide within its first five days of operation [2]. Thousands of ATMs for Bank of America and Washington Mutual were crippled; Continental Airlines flights were delayed or cancelled, the city of Seattle's emergency 911 network was knocked offline [3, 4], and most of South Korea's Internet services were blacked out for hours [5].

Mydoom is another fast spreading email worm [6]. A few hours after its first appearance, the average load time for Web pages increased by 50%. Only after two days, new variants of Mydoom emerged and the spread of the worm reached its peak: roughly one in five emails sent was infected by the worm. According to the security firm mi2g [7], by February 2004, Mydoom had caused $38.5 billion USD economic damage worldwide.

Traditional mechanisms to fight against these fast spreading worms rely on passive defense schemes. Unfortunately, these are often ill-coordinated stop-gap measures, for example, by forcing end-users to patch their systems regularly and updating antivirus software and firewall with latest virus patterns. These end-host based approaches have drawbacks such as slow response to new virus threats, high maintenance cost, significant reliance on human effort, and a disruptive update process. The inability to respond rapidly is lethal when new worms are designed to contaminate tens of thousands of hosts quickly such as ten minutes in the example of the Slammer worm. It is unrealistic to expect every end host in a large enterprise network to be patched within such a short time frame, not to mention that security software vendors also need a reasonable amount of time to develop patches.

Thus, a more effective approach against rapid worm spread is to stop worm propagation in the network before contaminated packets reach a significant number of end users. However, performing in-network packet inspection is not easy. To achieve this, we need *deep packet inspection techniques* that scan packet payloads to analyze the meaning and purpose of the network traffic to distinguish malicious packets from normal packets. These inspection techniques are not only useful for detecting and filtering packets containing worm signatures but are also required by other newly emerging edge network services such as high speed firewalls (for protecting end hosts from security attacks), HyperText Transfer Protocol (HTTP) load balancing (smartly redirecting packets to different servers based on their HTTP requests), and Extensible Markup Language (XML) processing (facilitating the sharing of data across different systems).

Unfortunately, this in-network deep packet inspection ability is not currently supported by routers and switches. In the current Internet, most of this kind of processing is restricted to the packet header, rather than the full packet content. It is hard to provide deep packet

inspection functionalities in the routers that run at the high line rates required of today's Internet due to the complexity of deep packet inspection systems.

The next two sections elaborate on why deep packet inspection is a hard problem: Section 1.2 reviews the current packet processing systems and explains why deep packet inspection ability is not present in the current Internet. Section 1.3 identifies the technical challenges that make it hard to incorporate such functionality in routers. The goal of this dissertation is to address these challenges and bring the high speed deep packet inspection techniques to the network. To keep up with packet processing in existing high-speed networks, we propose deep packet inspection schemes that are optimized for new hardware technologies such as Ternary Content Addressable Memory (TCAM) and multi-core processors. We present an overview of our work in Section 1.4. Finally in Section 1.5, we give the roadmap of this dissertation.

1.2 Today's Packet Processing Systems in the Internet

Today's Internet is largely built based on the famous "end-to-end" design principle [8] proposed by Saltzer, et al., which guides placement of functions among the modules of a distributed computer system. They observe that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level. Hence, they suggest that *specific application-level functions usually cannot, and preferably should not, be built into the lower levels of the system—the core of the network*.

Figure 1-1 shows that the End-to-end argument leads to dramatically different packet processing systems at different places of the Internet. Functionalities in the core are very simple, focused on forwarding packets. The complex functions related to application semantics are all built on end hosts. Next, we explain these kinds of packet processing

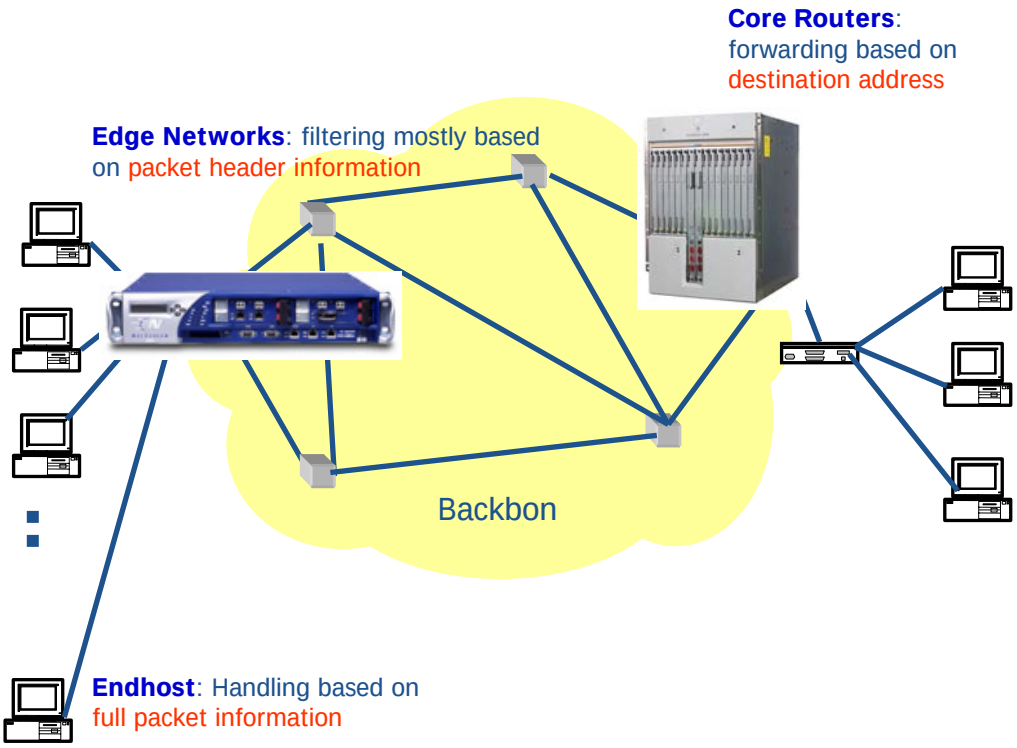


Figure 1-1. Current internet architecture.

systems by following the journey of a packet from an end host, to the edge network, then to core of the network.

An *end host* is a computer that directly interacts with users. It has access to full packet content information. Based on specific user application information such as the user's communication request (e.g., a Web browse request), and communication protocols, such as HTTP, or TCP, an end host computer packs application data into packets.

When a packet leaves the end host, it traverses the *edge network*, where all the packets sent by end hosts in this network gather together. Examples of such edge networks include campus networks and local area networks in large corporations. Such edge networks typically operate at Gigabit rates today. In these networks, packets are mostly processed based on the header information, and the result of the processing is typically only the next hop router to be used for forwarding this packet.

After a packet leaves the edge network, it can either enter another edge network, or more frequently, it travels to the core network of the Internet. In the core network, the packet processing rate gets even higher, 10 Gbps is quite normal and ISPs are beginning to deploy 40 Gbps links. Under such high speed requirements, the packet processing in the core must be very simple. Core routers do not touch the packet payload, sometimes not even the full packet header. Typically, core routers forward packets only based on one field in the packet header: the destination IP address.

This architecture, which has a simple core and complex end hosts, helps the Internet scale to 390 million hosts as of January 2006 [9]. However, this architecture also leaves security holes for the intruders to exploit. An intruder can disguise malicious packets that are piggybacked on known valid protocols and sneak through the network without being noticed. These packets can bypass the edge network inspection systems because they "appear" as acceptable packets with normal packet headers. They may then travel freely in the core network since they have valid destination IP addresses. Therefore, these malicious packets can easily reach other end hosts and infect those machines. When new machines are infected, new worms packets can be sent out to infect more end hosts. As a result, a worm can infect millions of machines quickly, thus causing service disruptions with enormous economic impact.

Recently, Network Intrusion Detection Systems (NIDS) have emerged that are built to filter out worms in the network. The core function of NIDS is to perform full packet (both packet header and packet payload) inspection against thousands of known virus or worm patterns. We refer to such full packet processing as *deep packet inspection*. Deep packet inspection is not supported by the routers in the Internet today. The major obstacle for incorporating deep packet inspection functionality into the network is its slow speed. Traditional processing algorithms are highly complex and their speed is well below Gigabits

process rate at the router. If current deep packet inspection modules are integrated into the network, it can easily become the performance bottleneck. In the next section, we study the challenges of performing high speed deep packet inspection at high rates.

1.3 Challenges for Deep Packet Inspection

Developing high speed deep packet systems is a challenging task because of the following factors.

- **Large number of signatures:** There are a wide variety of attacks and problems. One packet needs to be matched against thousands of attack signatures. For example, NIDS systems like the SNORT intrusion detection system [10] contains 4867 rules as of February 2006, each containing attack signatures.
- **Signatures have overlaps:** One packet maybe related to multiple attacks. For example, an HTTP packet can be vulnerable to 1096 different types of attacks known to the SNORT system. Hence, for a given packet, we first need to identify the related rules, and then check the packet contents against these signatures.
- **Patterns are often highly complex:** There are *fixed string patterns* with arbitrarily lengths; *correlated patterns* where it is necessary to check for certain groups of patterns occurring in sequence; and *patterns with negation* where it is required to detect the absence of a pattern. Sometimes, it is impossible to enumerate the pattern using a fixed list of strings, so *regular expressions* are used as a pattern language.
- **Location of signatures in the packet is unknown:** Due to the wide variety of application formats, we often do not know apriori the location of signatures in the packet payload. Hence, we need to check every byte of the packet payload at high speed.

We have shown that deep packet inspection systems are complex. To build such a complex system, there are two design elements that are essential: high speed and low cost, which we will explain below.

The key requirement for deep packet inspection system is *high speed* to match the normal processing speed of Internet core (10 Gbps, the common speed for core links) and edge networks (1 Gbps). We need to carefully inspect every incoming packet in both edge and core networks to see whether there is any attack attempt. In addition, worms and viruses may possibly originate inside the edge networks. Hence, we are also required to scan packets inside the networks, which often have internal connections in the gigabit rates or higher range. Therefore, deep packet inspection must be performed at Gigabit rates to build a practical in-network worm detection system. However, current deep packet inspection system cannot achieve such rates. The SNORT network instruction detection system [10], for example, implements packet inspection algorithms in software. It can handle link rates only up to 250Mbps [11] under normal traffic conditions even using SUN's special Security Defense Appliances [12]. In the worst case its performance is even lower. These rates are not sufficient to meet the needs of even medium-speed access or edge networks.

The second requirement is *low cost.* Many low performance related problems can be solved by using more computing power, e.g., more powerful CPUs, faster memory modules, multiple processors or computers. However, blindly increasing the amount of computation power increases both the hardware cost and also the maintenance cost. In addition, a large number of computers consume a lot of energy and impose high cost on the cooling system. Therefore, in this dissertation, we aim to develop smarter algorithms that use the smallest possible amount of processing power exploiting the hardware that can solve the problem most efficiently.

1.4 Focus of This Dissertation

The previous section showed that high speed deep packet inspection is a challenging task. We aim to address these challenges in this dissertation. In this section, we give an overview of our work. We first define the problem that we address in the area of deep packet inspection in Section 1.4.1. Deep packet inspection problem has several related problems, such as viruses and worm signature generation, packet reassembly, packet inspection. In this dissertation, we focus on the fast packet inspection problem and we list all the assumptions we make in Section 1.4.2. Next, in Section 1.4.3, we present our main technical components for deep packet inspections and state our contributions. Finally, we describe the evaluation metrics for our measurement methodology.

1.4.1 Evaluation Metrics

This dissertation addresses the problem of building high speed packet processing systems for services. We use the following evaluation metrics for the purpose of our study: the packet processing rate, memory requirements, power consumption, and the ease of incorporating new signatures.

- We define each in turn. *Packet processing rate* denotes the throughput of the system. Some of the schemes that we propose have a deterministic packet processing rate, while others are traffic sensitive. The reason for this is that the packet processing rate varies under different types of traffic patterns.
- For all cases, we report both the worst case throughput and average throughput. *Memory requirements* denote the size of memory used. Our solution leverages different types of memory for different operations, including Static Random Access Memory (SRAM), Ternary Content Addressable Memory (TCAM), and Dynamic Random Access Memory (DRAM). These memories have dramatically different

manufacturing costs, which in turn affect the total router cost. In our analysis, we report the memory requirements for each type of memories separately.

- Some of the special memory we use, such as TCAMs, consume large amounts of power. Hence, when we use it, we also report the *power consumed* by these devices.
- Lastly, given that new viruses and worms are continuously being developed, our deep packet inspection schemes must be able to accommodate new signatures easily. Hence, we analyze the *ease of incorporating new signatures*, which is defined as the time to process a signature and insert it into the system.

Now we have defined the problem and explained the evaluation metrics. Before we explain the components of our scheme in detail, we describe the assumptions we made in throughout the thesis in the following section.

1.4.2 Assumptions

Figure 1-2 shows that our deep packet inspection scheme processes incoming packets and compares them with a large number of signatures in parallel. We focus on developing Gigabit rate deep packet processing schemes. There are two assumptions we make in this architecture. First, we assume that signatures are known in advance. To ensure the signatures we use for evaluation are realistic, we obtain signatures from open source projects and do not attempt to generate synthetic signatures ourselves. The second assumption is that packets are defragmented first before entering our system. Next we explain the implications of these two assumptions in detail.

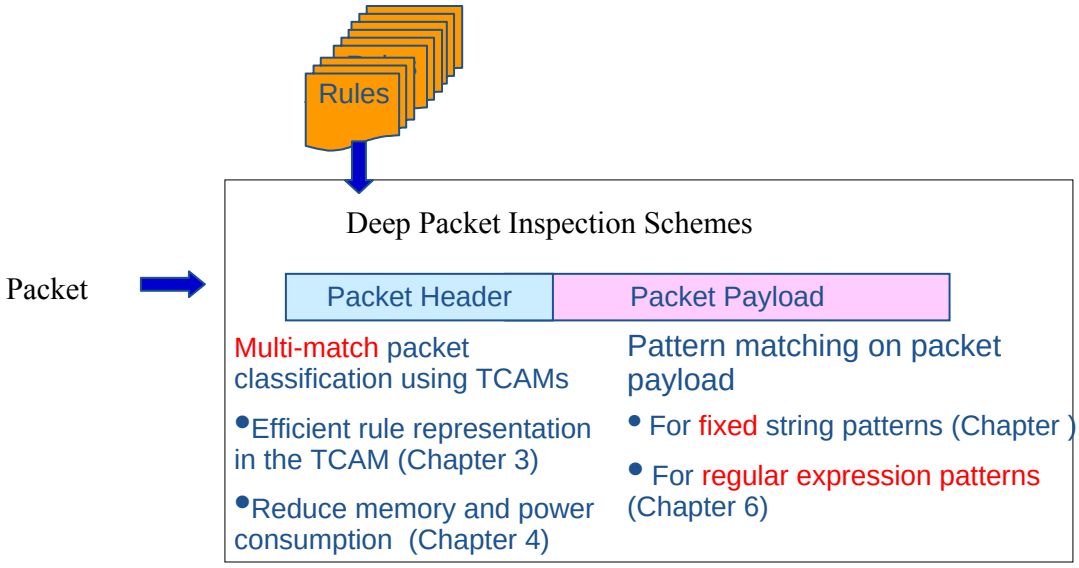


Figure 1-2. Deep packet inspection architecture.

Assumption 1: Signatures are known in advance

In our deep packet inspection system, incoming packets are compared against a set of traffic signatures. We obtain signatures from other systems in use in real-world environments. There are many research projects aiming to generate virus and worm signatures [13-15]. Software companies such as Microsoft and Symantec publish system vulnerabilities and corresponding signatures. Large networking companies like Cisco generate their own proprietary signatures. In addition, there are many open source projects aiming to generate signatures for detected worms [10] [16, 17]. For the experiments and studies throughout this dissertation, we obtain signatures from these open source projects. For example, the viruses and worm signatures used in our system are from the SNORT instruction detection systems [10], the Bro intrusion detection system [16], and the ClamAV virus signature database [17].

Assumption 2: Packet are Defragmented and Reassembled

The current Internet is a packet switching network. In most of its constituent sub-networks, such as Ethernet, there is a predefined maximum packet size that this network can carry. When a packet reaches a network, where the maximum packet size is larger than this packet's size, we need to fragment it into pieces, each small enough to pass over a link.

Often, there are multiple paths between the source and destination. Packets may take different routes. Hence, at the destination, packets can arrive in a different order from the order that was sent out. These out-of-order packets need to be buffered and then correctly ordered before we can get the correct application data.

The process of packet re-ordering and de-fragmentation to reconstruct the correct application data is called TCP reassembly. Packet reassembly is a hard task because we need to keep the states of millions of flows and we need to buffer all the out-of-order packets. There are existing systems that are built for packet reassembly [18-20] and the typical speeds reporting are 1 to 10 Gigabits rate under normal traffic. How to deal with worst case scenario, where many packets are fragmented and sent out of order, still remains a research topic. In this dissertation, we do not focus on the TCP reassembly problem. We assume packets are reassembled before analysis by our system.

Having explained the assumptions of our deep packet inspection system, next we explain the main components of the system and highlight our contributions.

1.4.3 Main Components and Contributions

The goal of this dissertation is to develop algorithms and schemes for high speed deep packet inspection. Accomplishing this goal requires two technical components: processing the packet header and processing the packet payload. In this dissertation, we separate these two because the packet header and the payload typically have significantly different

structures. The packet header is well structured: it has fixed fields with known offsets. So, we process it based on specific field information such as IP address, port number, protocol type, etc. We call this process *packet classification*. In contrast, the packet payload contains application data. Although a few application protocols, such as the HTTP protocol, have well-known formats, many other applications may make use of arbitrary formats with no standard rules. For this type of payload data, we need to perform *pattern matching* against thousands of patterns that can start from anywhere in the payload.

Note that packet payload scanning is also called Layer 7 processing according to the Open Systems Interconnection (OSI) reference model [21]. The OSI model has seven layers.

- Layers 1-2 are for physically transmitting the packets over wires, hence they are not of interest to this dissertation.
- Layers 3-4 are for sending packets over the inter-connected network. The packet classification problem may be applied in these two layers to scan packet *headers*.
- Layers 5 and 6 are the session layer and presentation layer, which are not widely used now.
- Layer 7 is the application layer, and as previously mentioned the packet classification problem may be applied at this layer to scan packet *payloads*.

Figure 1-2 illustrates the main technical components of our deep packet inspection system. For packet header processing, we propose a packet classification scheme that intelligently processes packet header information. For packet payload processing, we develop schemes to identify both fixed string patterns and complex regular expressions. These techniques form a cohesive architecture that can perform Gigabit rate packet scanning against thousands of sophisticated patterns. We explain these techniques in detail below.

Multi-match Packet Classification

As we mentioned before, rules in deep packet inspections may overlap with each other, so one packet may match multiple rules. We call the problem of identifying the related rules for one packet the multi-match classification problem. The challenge of multi-match classification lies in finding all the matching classification rules among thousands or tens of thousands rules within a few cycles. To address this challenge, we present our approach based on Ternary Content Addressable Memory (TCAM), which is a type of memory that can do fast parallel comparisons. Our approach produces multi-match classification results with only two memory lookups per packet. In addition, some filter sets use a *negation format* to represent "not-matching" (the absence of a match) of some specific values. Negations are not directly supported by TCAMs. Hence, we present an algorithm to remove the negations in the filter sets by translating them into equivalent operations the TCAM can easily handle.

TCAM is expensive and consumes a large amount of power. None of the previously published multi-match classification schemes is both memory and power efficient. Hence, we develop a novel scheme called the Set Splitting Algorithm (SSA) that meets both requirements. The main idea of SSA is to split filters into multiple groups and perform separate TCAM lookups in each of these groups. SSA guarantees a reduction in the number of filters by 50%, resulting in lower TCAM usage. In addition, it only accesses filters in the TCAM once per packet, yielding a 75% to 95% reduction in power consumption as compared to previously published schemes.

Fast Virus and Worm Signature Scanning with TCAM

The multi-match classification applies to the packet header only. For the packet payload, we need to check whether it contains any signatures. Applications such as network intrusion

detection require searching the packet payload against thousands of string patterns. We adopt TCAM for pattern matching functions because of its fast speed in parallel search. However, TCAM imposes physical limitations on the pattern length that can be directly matched. Also, TCAM does not handle correlated patterns (detecting whether a particular patterns occurs after another given pattern). To solve these problems, we develop algorithms that leverage TCAM's high speeds while not being restricted to these limitations. These algorithms can efficiently handle long patterns, correlated patterns, and patterns with negation. Our solution is also applicable to other Layer 7 pattern matching problems, for example, applications like HTTP load balancing and email SPAM filtering.

Fast Regular Expression Matching for Network Security Applications

Regular expressions are replacing fixed string patterns as the pattern matching language of choice in network security applications, such as SNORT, Bro, and the application layer packet classifier for Linux. Unfortunately, memory consumption is prohibitively high when traditional methods such as Deterministic Finite Automata (DFA) are used for fast regular expression scanning. To address this problem, we proposed rewrite techniques that can effectively reduce the size of DFA for highly complex regular expressions. Furthermore, we developed a scheme that can compile a large set of regular expressions into a small number of DFA, which dramatically improves the regular expression matching speed without significantly increasing memory usage. This scheme can leverage multi-core or Network Processing Unit (NPU) architectures to further improve performance. Experimental results using real-world traffic and patterns have shown that my implementation achieves one to three orders of magnitude speedup over the state-of-the-art implementation based on Non-Deterministic Finite Automata.

With our schemes that operate on both the packet header and packet payload, we can perform Gigabit rate packet scanning against thousands of sophisticated patterns. Using these techniques, we can obtain fine granularity analysis of the packets, and thus supporting new services such as network intrusion detection, high speed firewalls, HTTP load balancing, XML processing, etc., at high rates.

1.5 Dissertation Overview

The reminder of the thesis is organized as follows. Chapter 2 reviews work related to this dissertation. We broadly review the packet processing techniques that are presented at levels of the network, the repetitive deep packet inspection systems and show that pure-software based solution is insufficient to meet the need of today's network speed. At the end of Chapter 2, we present some emerging hardware technologies that can potentially speed up existing packet processing systems.

Chapters 3 to 6 are the main technical chapters. As mentioned previously, high speed deep packet inspection involves two technical components: processing the packet header and processing the packet payload. Chapters 3 and 4 address the first technical component – *multi-match packet classification*. Chapters 5 and 6 address the second component – *fast pattern matching* on the packet payload.

Chapter 3 presents a TCAM-based method for high speed multi-match classification. We present a *Geometric Intersection Method* that reports the multi-match classification results with just two lookups. In addition, we present a negation removing algorithm that can efficiently map rules into the TCAM.

The multi-match classification solution presented in Chapter 3 is extremely fast. However, for some applications, it will use a large amount of memory and also consume a

large amount of power. In Chapter 4, we develop a general and balanced scheme that takes into consideration classification speed, memory consumption and power consumption.

Chapters 5 and 6 present schemes for fast pattern matching on packet payload. In Chapter 5, we study the typical patterns in the packet payload scanning applications and define three types of patterns: fixed string patterns, composite string patterns and regular expression patterns. We address the first two types of patterns in Chapter 5 and present a TCAM-based string searching algorithm that can operate at gigabit rates.

Chapter 6 deals with the complex regular expression patterns. Through studying the patterns, we show that some either need a significant amount of memory or have a high computation cost. We propose regular expression techniques that rewrite these patterns into one that are more memory efficient. Further, we present a fast single-pass scanning algorithm for simultaneously scanning thousands of patterns. We implement a fast and memory efficient regular expression scanner for real-world patterns of packet scanning applications. We demonstrate the effectiveness of our scanner through comparing it with different packet scanning solutions used in current packet scanning applications.

Chapter 7 concludes the dissertation and discusses several directions for future work. The main contribution of this dissertation lies in using emerging hardware technologies for fast packet processing. In Chapters 3 to 5, we adopt TCAMs for fast processing due to its building parallel comparison ability. In Chapter 6, we present algorithms that are suited for both single core and multi-core processors. Our work can be extended to other hardware technologies such as FPGA and ASIC-based platforms.

2 Background

The previous chapter showed that we need deep packet inspection systems to stop fast-propagating viruses and worms. These systems typically have a large rule database and they compare the packet against these. Due to the complexity of rules, existing systems are all software-based and the packet processing speed is very limited. In this section, we first review these systems and taxonomize their requirements in Section 2.1. Then in Section 2.2, we review the existing deep packet inspection methods and show that none of them can operate at Gigabit rates, given thousands of complex signatures. Finally, in Section 2.3, we present some emerging hardware technologies that can potentially boost the deep packet inspection systems to Gigabit rates.

2.1 Packet Scanning Systems

There are many packet scanning applications that require deep packet inspections. Here, we review three popular ones: SNORT [10], Bro [16] and Linux L7-filter [22]. SNORT and Bro are two popular intrusion detections systems, while Linux L7-filter is for application protocol analysis. These systems are all open source systems, which allow us to perform a detailed analysis and show their abilities and constraints.

2.1.1 SNORT

SNORT is a popular open source intrusion detection system, with millions of downloads to date [10]. It can be configured to perform protocol analysis and content searching and matching on real-time traffic to detect a variety of worms, attacks and probes.

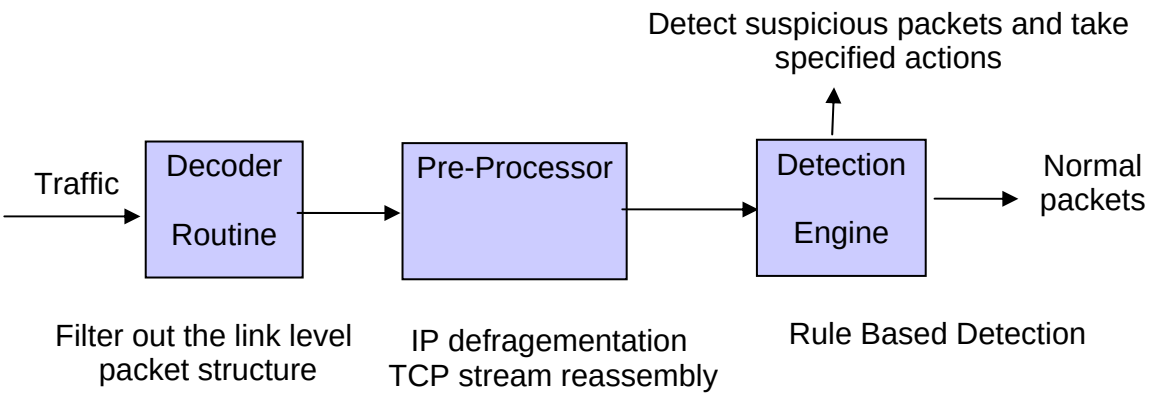


Figure 2-1. SNORT architecture.

Figure 2-1 illustrates the flowchart of the SNORT system. When a packet enters the network, it enters the decoder. Here the link level information, such as the Ethernet packet header, is removed. Then, the packet enters the pre-processor, which performs packet defragmentation and reassembles the TCP stream. Finally and most importantly, the packet enters the detection engine. The detection engine examines that data against its database of rule. If the packet matches a rule then SNORT takes a pre-programmed action, e.g., "drop this packet", or "log this packet".

SNORT contains thousands of rules, each containing attack signatures. Each rule in SNORT has two components: a *rule header* and a *rule option*. The rule header is a classification rule that applies to the packet header. This rule header consists of five fixed fields: protocol, source IP, source port, destination IP, and destination port. Figure 2-2.a gives an example SNORT rule that detects a MS-SQL worm probe. Here, the rule header specifies that this rule applied to User Datagram Protocol (UDP) packet from external network to a computer in the protected network (called home net) with port 1434. Figure 2-2.b is another example rule for detecting an RPC old password overflow attempt. It applies to any UDP packet from the external network to a computer in the home network. The rule headers in these two examples overlap with each other, so a packet may match both rule headers and

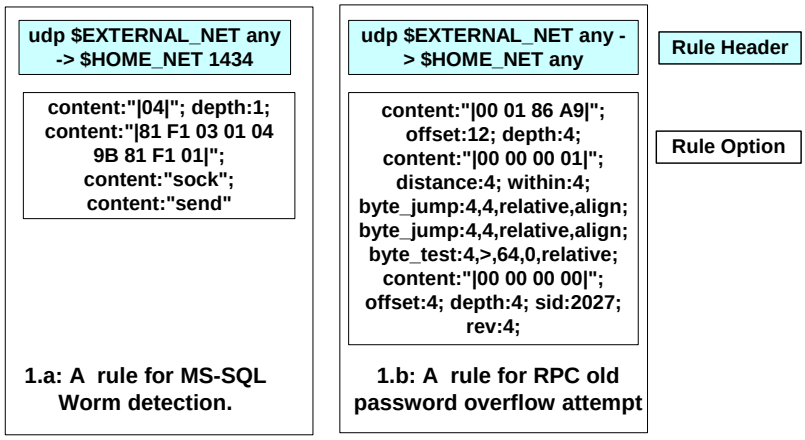


Figure 2-2. SNORT rule examples.

need to be checked against both rule options. In the SNORT systems, a packet may match thousands of rule headers.

The rule option is more complicated: it specifies which intrusion patterns are to be used to scan packet payload. As we defined in Section 1.3, there are four types of patterns: fixed string patterns, correlated patterns, patterns with negation and full regular expression patterns. The SNORT system has rules of all four pattern types. For example, the MS-SQL worm detection rule (Figure 2-2.a) requires a sequential matching of a correlated pattern that consists of four string patterns. The RPC worm detection rule (Figure 2-2.b) searches for a pattern with negation. After matching a pattern *"USER"*, if it does not detect a return character (*\n, |0a|* in ASCII format) within the next *50* bytes, it will raise an intrusion alarm signaling an overflow attack attempt. Recently, SNORT also incorporates a large number of regular expression patterns. For example, the pattern for detecting Internet Message Access Protocol (IMAP) email server buffer overflow attacks is ".**AUTH*\s[^\n]*{100}*". This signature detects the case where there are 100 non-return characters "[^\n]" after matching of keyword *AUTH*\s. Matching of these signatures is the core component of the SNORT system. After introducing the rule structure in SNORT, next we explain the string matching algorithms used in SNORT.

For the string pattern matching, SNORT uses a "parallel Boyer-Moore" approach that has been explored in the literature for fast matching of multiple strings. The original Boyer-Moore algorithms are designed for single pattern searching [23]. They build skip tables to avoid back tracking and help shift forward. The search time for an *m* byte pattern in an *n* byte packet is *O(n+m)*. If there are *k* patterns, the search time is *O(k(n+m))* , which grows linearly in *k*. Hence, this method is slow when there are thousands of patterns. The parallel Boyer-Moore algorithm used in SNORT can potentially decrease the running time to sub-linear time in *k* for certain packets. However, this performance is not guaranteed, and for some packets it requires super-linear in *k* time to perform matching [24]. Recently, new pattern matching algorithms are proposed to boost the pattern matching speed of SNORT. For example, the Aho-Corasick-Boyer-Moore (AC_BM) algorithm proposed by Silicon Defense [25] combines the Boyer-Moore and Aho-Corasick algorithms. Another new algorithm is the Setwise Boyer-Moore-Horspool algorithm by Fish, et al. [26], whose average case performance is better than Aho-Corasick and Boyer-Moore. These algorithms greatly improve SNORT's pattern matching speed to a certain level, e.g., 250Mbps with the SUN SDA[12]. However, it is still below the line rate needed for network deployment.

2.1.2 Bro

Bro [16] is an open-source intrusion detection system developed by Vern Paxson, et al. Similar to SNORT, it monitors network traffic and detects intrusions by comparing network traffic against a set of rules describing attack signatures. In addition, it records unusual activities such as a high number of failed connection attempts. When Bro detects a suspicious event, it performs a set of pre-configured actions, such as logging the event, terminating the connection, and sending a message to system administrators.

Different from SNORT, Bro does not use fixed string patterns. Instead, all the patterns are expressed using regular expressions. Figure 2-3 is an example rule for detecting the Code Red worm [27]. This rule applies to tcp packets that direct to http_ports (e.g., port 80), with an established tcp connection, aiming to detect a HTTP request matching the regular expression *"/.*\.id[aq]\?.*NNNNNNNNNNNN"*. This regular expression denotes finding a pattern that can start from anywhere *(.*)* in the HTTP request, starting with a string *".id"*, followed by character *"a"* or *"q"* and a quotation mark character *"\?"*, then some arbitrary characters of any length *(.*)*, finally end with the string *"NNNNNNNNNNNN"*.

```
Signature codered1 {
        ip-proto == tcp
        dst-port == http_ports
        tcp-state established,originator
        http /.*\.id[aq]\?.*NNNNNNNNNNNN/
        event "CodeRed 1"
}
```

Figure 2-3. An example Bro rule for detecting the Code Red worm.

Bro uses a Deterministic Finite Automaton (DFA) based approach for regular expression matching. A DFA consists of a finite set of states, and a transition function. At any time there is only one active state in the DFA. Given the next input character, DFA jumps to the next state following the transition function. Hence, DFA has a deterministic throughput regardless of the input. In addition, if we compile several regular expressions into a single DFA, the processing cost for one input character remains the same. This nice property helps Bro process thousands of patterns quickly.

However, some regular expressions can cause a DFA to grow exponentially so that the states cannot fit in the memory. To solve this problem, Bro adopts a lazy DFA-based approach, where commonly used DFA states are cached and the DFA is extended at run-time if needed. This lazy DFA-based approach, although fast and memory efficient on most

common inputs, may be exploited by intruders to construct malicious packets that force the lazy DFA to enter many corner cases [24] which result in drastically reduced performance.

2.1.3 Application Layer Packet Classifier for Linux (Linux L7-filter)

We have shown two intrusion detection systems, SNORT and Bro that aim to identify *malicious* packets. In this section, we introduce another type of packet monitoring system – Linux L7-filter, whose goal is to identify the application protocol information of *normal* packets.

The goal of the Linux L7-filter is to detect the application layer protocols. The examples of application layer protocols are Yahoo messenger protocol, Peer-to-peer (P2P) Kazza protocol, and online gaming protocol. Identifying these types of application traffic is important for providing fine granularities of quality of service. For example, according to the network infrastructure company CacheLogic, in 2004 over 60 percent of all Internet traffic is P2P traffic, such as Kazza and BitTorrent traffic. Most of these P2P traffic is illegal data transfer. When this P2P traffic occupies too much network bandwidth and causes network congestion, service providers may need to limit the bandwidth of P2P traffic so as to provide a better quality of service (QoS) for more important traffic such as routing HTTP and Voice over IP (VoIP) packets. To do this, it becomes necessary to distinguish P2P traffic from other types of traffic. The Linux L7-filter can help us achieve this through analyzing application layer protocol information.

Currently, the Linux L7-filter contains 70 application protocol signatures, contributed by researchers and developers world-wide. Signatures are categorized into ten classes: P2P, VoIP, streaming video, streaming audio, chat, game, networking, mail, file transfer, printer commands. Similar to Bro, signatures are specified using regular expressions. However, different from SNORT and Bro, the Linux L7-filter does not look at the packet header

information such as port number because different some applications can be piggybacked on known applications protocols. Therefore, the Linux L7-filter closely checks application layer data to determine what protocol is being used.

The Linux L7-filter is a pure software solution, it cannot achieve Gigabit rate. We found that the system throughput dropped to less than 10Mbps on a 100Mbps network using a computer with 750MHz PIII and 512 MB of memory. In addition, the CPU was saturated, leaving no computation time free for other tasks. Apparently, this speed does not satisfy the requirements for high speed edge networks, where Gigabit Ethernets have become the norm.

So far, we have shown three typical deep packet inspection applications. These applications are highly complex, so currently they rely on purely software-based approaches that are designed to run on standalone computers such as Linux boxes. The packet processing rates of these systems are much lower than Gigabit rate. To incorporate them into the network, we need to increase the speed to match the high processing rate in the edge and core networks. In the next section, we broadly review the current packet processing systems on routers and show how most current packet processing systems can meet our needs. Later in Section 2.3, we present some emerging hardware opportunities that can potentially boost these packet scanning systems to Gigabit rates or higher.

2.2 Current Packet Processing Approaches

In this section, we broadly review the state-of-the-art packet processing technologies. As we mentioned in Section 1.4, deep packet inspection involves processing both the packet header and the packet payload. In this section, we first review packet header processing techniques and then we review packet payload processing methods that look beyond the packet header. In both parts, we review both algorithms developed by researchers as well as several commercial products.

2.2.1 Packet Header Processing

Packet headers are typically well structured, following the pre-defined packet header formats, making processing a simpler task. Packet header processing is usually targeted towards matching specific fields in the packet header. For example, in the core of the Internet, packet forwarding is usually just based on one field — the destination IP address. Some edge routers control traffic using more fields in the packet header. Typical rules often reference the protocol, source IP address, destination address, source port number, destination port number fields of the packet. Cisco allows filtering based on these five tuples by configuration of Access Control Lists (ACLs).

The packet classification problem on multiple fields is a complex problem [28]. For N arbitrary non-overlapping regions in F dimensions, it is possible to achieve an optimized query time of $O(log(N))$, with a complexity of $O(N^F)$ storage in the theoretical worst case [29]. This result can be directly mapped to packet classification applications, where a filter with F fields corresponds to a region in the F dimensional space [29, 30]. Suppose we have $N = 1000$ filters and each filter has five fields ($F = 5$). To achieve an optimal query time $O(\log N) = 10$ memory accesses, we need on the order of $N^F = 100$ terabytes of storage. On the other hand, to achieve an optimal storage of $O(N)$, we need on the order of $log^{F-1}N = 9840$ memory accesses. Fortunately, these analyses represent a theoretical worst case; real-world rule sets are typically simpler than the theoretical worst case. Many algorithms have been proposed to solve the multiple filed packet classification problem. In this section, we list several representatives. These include trie-based, hash-based, and heuristic algorithms. At the end of this section, we review the industry approaches and state their limitations in providing a solution for deep packet inspection.

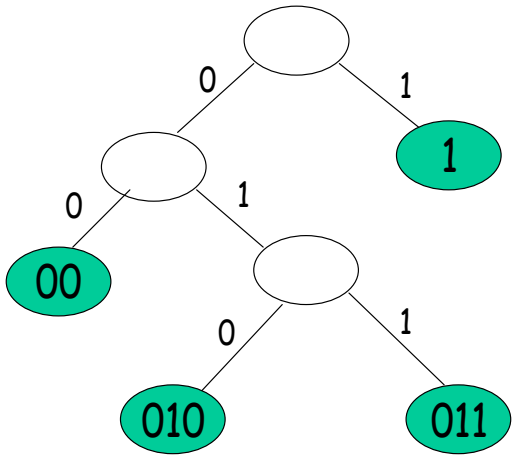


Figure 2-4. An example of Patricia Tree for storing strings 1, 00, 010, and 011.

Trie-based Algorithms:

Trie-based solutions are the most traditional solution for packet classification. A trie is a general-purpose data structure for storing strings as shown in Figure 2-4. The IP address may be represented as a string of bits, so we can use trie-based approach to identify IP addresses. The most well known trie for packet classification is the Patricia Trie [31]. Patricia tries store prefixes in a tree-like structure with a *0* and *1* on each link corresponding to the bit at the given level in the tree. In the worst case, to obtain the final classification results, we need as many memory accesses as the number of bits in the fields *(w)*. This example is for the classification for one field. For classification problems with multiple fields, Grid-of-tries [32] builds multiple levels of tries, each level corresponds to a field. Once a match is found in one dimension, a search is performed in a second level trie tree linked by the first level tree node. Another trie-based approach, Area-based Quad Tree (AQT) [33] works for two dimensional space. It keeps dividing separating the space into four regions recursively until the number of filters in each region is smaller than a constant.

In general, trie-based schemes do not work very well at multiple-field classification because the number of memory lookups is related with the number bits in all fields. For the

popular five field classification based on protocol, source IP address, destination address, source port number, destination port number, there are a total of 104 bits and hence in a worst case 104 memory lookups is needed.

Hash-based Algorithms (Tuple based algorithms):

Trie based algorithms are built purely based on the classification rules. They do not take into consideration of the incoming packets' distributions. Hash-based algorithms, however, do take this information into consideration. Hash based algorithms perform a series of hash lookups for each possible prefix length to identify the highest priority matching rule. Hash-based algorithms, in the worst case, require memory accesses to all of the hash tables and thus need to conduct many memory accesses. However, the hash function can be optimized based on the distribution of incoming packets. One can develop good hash functions that require one memory access for the most common prefixes.

Srinivasan, et al., observed that in real applications, the filter database typically uses only a small number of distinct field lengths. Therefore, they proposed that by mapping filters to tuples, even a simple linear search of the tuple space (one hash probe for each tuple space) can provide a significant speedup against linear search [34]. They also demonstrated an optimized hashing technique for two dimensional spaces, called rectangle search. However, to support lookups in more than two dimensions, the algorithm still requires at least $W^{F-1}/F!$ memory accesses, where W is the number of bits in each field and F is the number of fields. Suppose $W = 16$ and $F = 5$, this yields 546 memory accesses.

Heuristic Algorithms:

Beside trie-based algorithms and hash based algorithms, there are heuristic algorithms that work great on real-world filter sets. Here, we discuss two heuristic algorithm examples: HiCuts [35] and HyperCuts [36]. The HiCuts algorithm works by carefully pre-processing

the classifier to build a decision tree data structure. Given a packet, the decision tree is traversed to find a leaf node, where a small number of rules are stored. A linear search among these rules yields the classification result. HyperCuts is another decision tree based algorithm. Unlike HiCuts, however, in which each node in the decision tree represents a hyper plane, each node in the HyperCuts decision tree represents a multi-dimensional hypercube. Using this extra degree of freedom and a new set of heuristics to find optimal hyper cubes for a given amount of storage, HyperCuts can provide an order of magnitude improvement over HiCuts classification algorithms. These heuristic approaches yield state-of-art best results, e.g., 20-30 memory accesses per packet in the "worst case".

We have reviewed representative algorithms developed by researchers for packet header processing. Next, we review the state of art commercial packet header processing systems.

Commercial Packet header Processing Systems

Commercial systems usually use caches to improve performance. For packet header processing, common packet headers can be cached to speed up future lookups. The cache hit rate for caching full IP addresses in routers is at most 80-90% [37, 38]; cache hit rate is likely to be much worse for caching full headers. Incurring a linear search cost to search through 100,000 filters would be a system bottleneck even if it occurs on only 10% to 20% of the packets [39].

When there is a cache miss, there are two choices. The first is to use *software-based algorithms* mentioned previously to identify the classification results. Since software-based algorithms are slow, some high end routers use *hardware based solutions* with Ternary Content Addressable Memory (TCAM). Here we give a brief description of TCAM and we will give a more detailed analysis of TCAM later in Section 2.3.2. TCAM is a type of special memory built for parallel searching. It has many entries and it can compare the input with all

its entries in parallel. Using TCAM for packet classification is very simple: just insert all the classification rules into the TCAM, one rule per TCAM entry. Given an input packet, TCAMs can compare the input with all the entries in parallel and report the classification result with just one TCAM lookup time (e.g., 4 ns), removing the uncertainty of most software-based solutions. Large routers such as Cisco 7304 MSC [40] use TCAM to support 10 Gigabit classification rates.

Limitation of Current Packet Classification Problem

All the Layer 4 classification solutions presented above are for single matching classification, i.e., they only report the highest priority match. As we have discussed in the Section 1.3, rules in intrusion detection systems often overlap with each other, and demand all the matching results (multi-match classification results). The multi-match classification problem is more complex to implement than single-match classification since it needs to return more than one matching result. Some of the heuristic software solutions for single match classification are not applicable to the multi-match problem. In addition, the hardware solution using TCAMs cannot be used to return multi-match classification results directly since TCAMs only return the first match result. In this dissertation, we particularly focus on algorithms for the multi-match classification problem.

2.2.2 Packet Payload Inspections

Now, we have surveyed packet header processing systems, next we survey the packet payload processing systems. Packet payload processing is different from the packet header processing. Packet headers have fixed formats, but packet payload may follow many different application formats. Often we do not know where the patterns appear. Therefore, we have to search entire packet payload.

High speed payload processing is required by many deep packet inspection systems such as SNORT, Bro, and Linux L7-filter. These systems typically have a large rule database and they compare incoming packets against all of these rules. Due to the complexity of rules, existing systems are all software-based and packet processing speed is very limited, as we have shown in Section 2.1.

Router vendors are also introducing new products for Layer 7 deep packet inspection. Some of the simple functions are supported by hardware, while most of the complicated rule matching is still done in software. Next we describe several commercial products.

PMC Sierra introduced the PM2329 ClassiPI network classification processor that supports 16K policy rules [41]. These rules can be spread over multiple independent engines for rule comparisons. Rules in one engine are searched sequentially. Rules can be written using regular expressions. However, the classification processor supports only a limited number of regular expressions (12 reported in [41]).

Cisco's Internetwork Operating Systems (IOS)-based Intrusion Prevention System (IPS) contains several Signature Micro Engines (SMEs) [42]. Each SME typically corresponds to the protocol in which the signature occurs and looks for malicious activities in that protocol. Given a packet, there may be multiple rules associated with it so it is processed by several SMEs. When an SME scans the packets, it extracts certain values and searches for patterns within the packet via the regular expression software. Similar to PCM Sierra's PM2329 ClassiPI, most of the inspections in Cisco IOS are software-based. There is no public report of the processing power of these systems when the number of rules is large.

These existing packet payload scanning systems have limitations on the number of rules, the type of regular expressions, and the number of bits in the packet. As the number of rules are getting larger (e.g., SNORT has more than 4000 rules) and more complex (different types of signatures are used), these existing software solutions is not sufficient and do not support

high processing rates. Instead, we need stronger hardware support to boost the performance to Gigabit rates. In the next section, we review the emerging hardware technologies that can potentially help us solve the deep packet inspection problems with high rates.

2.3 Hardware Opportunities

Hardware technologies are developing fast. To speed up deep packet inspection, we need hardware that has the following three characteristics: *a high degree of parallelism*, *fast processing time, low power consumption, and configurable*. In this section, we first explain these three requirements in detail. Then we list several technologically feasible hardware components that meet these, and thus achieve the necessary speedup for the deep packet inspection systems.

2.3.1 Requirements for Hardware Technologies

Deep packet inspection will compare a packet with thousands of signatures. We stated in Section 1.3 that this inspection process has the following characteristics. First, we must handle a large number of signatures. Second, these signatures are highly complex and have overlaps. Lastly, signatures change over time. In response to these characteristics, hardware technologies must meet the following requirements.

To support a large number of signatures, hardware technology for deep packet inspection must have *a high level of parallelism*. Any solution that requires comparing the packet with signatures one by one does not scale well. Second, hardware technology must have a *fast processing time* to process complex signatures to keep up with the Gigabit line rate. Since we must inspect packets from every line card, deep packet inspection solutions must use a *moderate amount of power* so as not to impose high load on the cooling system. Lastly, hardware must be *configurable*. New signatures will be developed for detecting new worms.

Or even new signature languages may be introduced in the future. Hence, it must allow the incorporation of these new signatures through flexible configuration.

To meet the above requirements, we identify the following three types of hardware that can be potentially used for high speed deep packet inspection. They are TCAM, FPGA, and Multi-core processors.

2.3.2 TCAM

TCAMs are a piece of specialized hardware built for parallel comparison. A TCAM consists of a large array of comparators which can perform parallel matching of its constituent entries. The top entry of the TCAM has the smallest index and the bottom entry has the largest. Each entry has several cells that can be used to store a string. A TCAM works as follows: given an input string, it compares this string against all entries in its memory in parallel, and reports one entry that matches the input. A TCAM has the following nice properties that make it an excellent choice for deep packet inspection.

1. **High degrees of parallelism.** TCAMs can be used to store a high number of entries. Single-chip densities of TCAMs are approaching 2 Megabytes (MB) [43]. The width of each entry can be configured according to user requirements. For example, a 1 MB TCAM can be programmed as 64K entries with 16 bytes per entry, or 1K entry with 1K bytes per entry, etc. Given an input string of 16 bytes, a 1MB TCAM can compare the string with all 64K entries in parallel.
2. **Efficient support for IP addresses.** Unlike a binary CAM, which only has two states: 0 or 1, each cell in a TCAM can take one of the three states: 0, 1, or '?' (do not care). With the 'do not care' state, TCAM can be used for matching variable prefix CIDR IP addresses and thus can be used in high-speed IP lookups [44]. There is a 1-to-1 correspondence between the TCAM memory bits and the number of bits in a

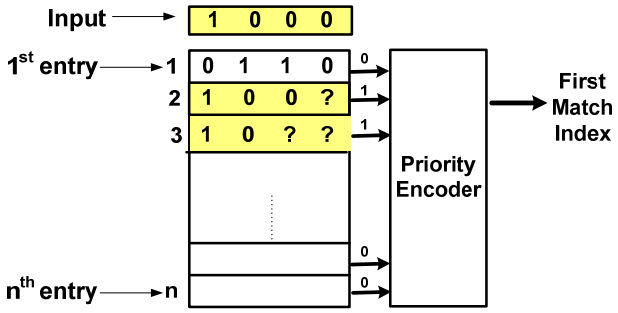


Figure 2-5. TCAM.

classification rule. This is the total number of bits in all fields. For example, an IP address has 32 bits and will hence uses 32 TCAM cells.

3. **Fast and deterministic lookup time.** The lookup time (e.g., 4 ns [43]) of TCAMs is very short and it is deterministic for any input. In the current Internet, the average packet size is 402.7 bytes [45]. If one packet requires one TCAM lookup for packet header processing, TCAM-based solution can achieve an 805.4 Gbps classification rate.

The three properties above make TCAM one of the top candidates for a deep packet inspection system. However, TCAMs also have limitations. Current off-the-shelf TCAMs can only report one of the matching results, even though there may have been many matches. This limits TCAMs' matching power in the presence of deep packet inspection rules that overlap. In addition, TCAMs are expensive and consume a lot of power. Next, we explain these limitations one by one.

- **Only report the highest match results.** Because of the 'do not care' state, one input may match multiple TCAM entries. Today's off-the-shelf TCAMs are first-match TCAM, which gives out the lowest index match of the input string if there are multiple matches. Figure 2-5 shows an example TCAM. If we remove the priority encoder, a TCAM returns a matching vector. The processor needs to step

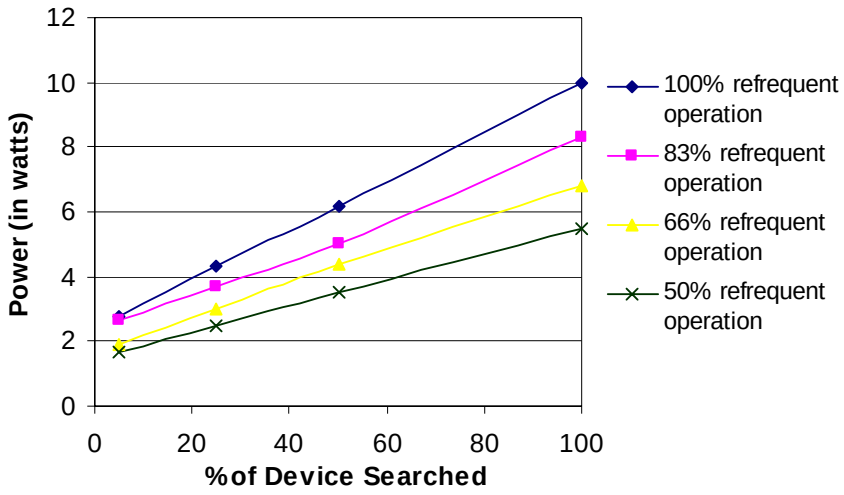


Figure 2-6. Power consumption for a 9 Mibt TCAM.

through the vector to extract the matching results. This is not computationally efficient when the number of entries in the TCAM is large and the matching vector is sparse, which is true for SNORT filter sets as we will show later in Section 4.4. In this dissertation, we use the off-the-shelf TCAMs and do not make changes to the priority encoder.

- **High cost.** TCAMs are more expensive than Static Random Access Memories (SRAMs). Table 2-1 shows that TCAMs cost about 30 times more per bit of storage than DDR SRAMs [44].
- **High power consumption.** TCAMs consume 150 times more power per bit than SRAMs [46]. Figure 2-6 shows the power consumption of a 9 Mibt TCAM. It is linear to number of entries searched in parallel, and it is also directly related to the frequency of TCAM access. Therefore, to develop a power efficient TCAM-based solution, we need to insert limited number of entries into the TCAM and access it as infrequent as possible.

To decrease the power consumption of TCAMs, TCAM vendors provide a *block* feature. A TCAM block is a contiguous, fixed-sized chunk of TCAM

entries, usually much smaller than the size of the entire TCAM [47]. For example, a 128K entry TCAM could be divided into 8 blocks containing 16K entries each. With this feature, we can selectively search one or several blocks, instead of the entire TCAMs, saving the energy consumption.

- **Low flexibility.** Each cell in TCAMs can take one of the three states: 0, 1 or "don't care". It can't support range matching directly. A filter containing range may require multiple TCAM entries e.g., port *2-5* is represented as *01** and *10**. In addition, TCAMs cannot support negation form, for example "not *4*". "not *4*" is essentially two ranges: *0-3* and *5* to the maximum value.

Table 2-1. Different memory technologies.

Technology	Single chip density	$/chip ($/MByte)	Access speed	Watts/chip
Networking DRAM	64 MB	$30-$50 ($0.50-$0.75)	40-80 ns	0.5-2W
SRAM	4 MB	$20-$30 ($5-$8)	4-8 ns	1-3W
TCAM	1 MB	$200-$250 ($200-$250)	4-8 ns	15-30W

As shown above, TCAMs have many nice properties, including massively parallel comparison abilities. However, we also showed that they have several limitations. In this dissertation, we develop algorithms that use TCAMs to achieve high speeds while not being restricted to these limitations.

2.3.3 FPGA

Similar to TCAMs, Field Programmable Gate Arrays (FPGAs) can also support a high degree of parallelism. FPGAs are semiconductor devices containing programmable logic components and programmable interconnects [48]. Engineers can program the logic components to perform different types of tasks such as AND, OR, XOR, NOT. These logic

components can be linked together through programmable interconnects to build complex functions such as data calculation.

FPGAs are an excellent candidate for deep packet inspection due to their abilities of parallelism, easy configuration, and high efficiency.

- **Support high level of parallelism.** The programmable logics in FPGAs can run in parallel, hence FPGAs can perform multiple tasks simultaneously.
- **Easy configuration.** The logic blocks and interconnects in FPGAs can be programmed after the manufacturing process. When there are new signatures or new detection architecture change, we can modify the logic blocks and interconnects to gain the desired functionalities in the FPGA.
- **High efficiency.** Deep packet inspection has quite different requirements from other applications such as 3D animation and scientific calculations. It mainly requires *logical comparisons*, while complex *numeric calculations* such as floating point number processing are rarely used. For FPGA-based approaches, we can customize the chip to provide the optimal performance for the required functions only.

The above attractive properties make FPGAs an excellent candidate for deep packet inspection systems. In fact, FPGAs have already been used for regular expression matching [49]. For example, Lockwood, et al., proposed a FPGA based solutions are used to detect worms and malware that scan traffic at rates up to 600Mbps [50]. Moreover, FPGAs are a multi-billion dollar industry, which leads to continous investments and advaces. TCAMs are special products for switch companies.

Comparing FPGAs to TCAMs, FPGA are more flexible and can handle more complex tasks. For example, one can build circuits for regular expression matching easily in FPGAs, which is difficult to support in TCAM. Hence, FPGAs are ideal to handle complex deep

packet inspection. However, for simple task such as straight forward string comparisons, such as packet classification, TCAMs are still a better choice than FPGA because TCAMs are a piece of hardware specially built for parallel comparison, it can run at higher frequencies and yield lower cost compared to FPGAs.

2.3.4 Multi-core Processors

We have explained that FPGAs are more flexible than TCAMs. General processors are even more flexible than FPGAs. We can program it using high level languages (e.g., C, C++) that are simpler to program, rather than the low level VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) for FPGAs which is relatively harder to master.

Recently, multi-core processors are becoming popular. Different from the traditional single core processors, multi-core processes combine two or more independent processors into a single package. These independent processors can run in parallel hence can provide higher computation power. As the processing power of single core processors are approaching its limit [51], multi-core processors are becoming popular to achieve higher computation power. For example, many PC vendors like Dell, IBM, and Apple, are selling PCs with dual-core processors. Networking equipment vendors are also using multi-core processors. The widely used Intel Network Processor Units (NPUs) have 8-16 cores [52]. Recently, IBM introduced cell processors with eight cores for fast processing pixels for video games [53]. Cisco also built a Silicon Packet Processor (SPP) that contains 188 32-bit embedded RISC cores for Cisco's high end routers [107].

Multi-core processors are a good candidate for high speed deep packet inspection because they have multiple cores that can provide parallel computations. In addition, they are

very flexible. We can easily port software approaches such as SNORT to multi-core environments. However, multi-core processors also have the following two limitations.

- **The number of cores is limited.** For example, Intel IXP2800 NPU [52] has 16 cores, which is much smaller than the number of patterns as it's typical to have thousands of patterns in deep packet inspections systems. Hence, one pattern per processing unit is infeasible. We need smart algorithms to partition different tasks and patterns into the different cores.
- **The size of fast local memory of each processing unit is limited.** For example, the newly architected IBM cell processor has 8 synergistic processor elements, each with 128 KB local memory [53].

2.3.5 Summaries on Hardware Technologies

We described three types of hardware technologies that are potentially useful for deep packet inspection: TCAMs, FPGAs, and multi-core processors. TCAMs are very specialized devices built just for parallel comparison. TCAMs work most efficiently, in terms of cost and speed, for tasks that need massive comparison of input with limited length strings. FPGAs on the other hand are more flexible, and can support more complex tasks. Multi-core processors are even more flexible, can support an even wider variety of computation tasks. However, for simple computation, TCAMs are sufficient.

In the next four chapters, we present our algorithms for deep packet inspections with these hardware technologies. For tasks that require a massive amount of simple comparisons, such as packet classification, we adopt TCAMs. As we mentioned earlier, TCAMs have four limitations: not being able to report all matching results, inflexible to support ranges and negations, have high costs, and consume a lot of power. We will present our TCAM-based schemes that address these limitations. Chapter 3 describes a mechanism for mapping

negations efficiently into the TCAMs, and presents an algorithm to report all the matching results with just one TCAM lookup for packet classifications. Chapter 4 extends Chapter 3, focusing on limiting memory and power consumption of the TCAM.

Beside packet header processing, deep packet inspection also requires payload processing, which compares the payload with a set of signatures. For fixed string-based signature comparison, we propose a TCAM-based solution in Chapter 5. Unfortunately, it is an extremely difficult problem to directly support regular expression based signatures in TCAMs. Hence, we propose algorithms for general processor and multi-core based architectures in Chapter 6. Our schemes can be extended to FPGA-based platforms [54].

3 Multi-Match Packet Classification with TCAM

As introduced in Chapter 1, the goal of this dissertation is to develop algorithms and system organization for high speed deep packet inspection. Accomplishing this requires two technical components: processing the packet header, and processing the packet payload as we explained in Section 1.5. In this chapter, we describe our approach for the first component, namely, multi-match packet classification that applies to the packet header. In particular, we present a scheme based on Ternary Content Addressable Memory (TCAM), which produces multi-match classification results with only two memory lookups per packet.

The rest of the chapter is organized as follows. We first motivate the problem in Section 3.1. Then we investigate the characteristics of multi-match classification sets in Section 3.2 and show that software-based single-match classification approaches do not work efficiently on the multi-match classification sets: they either need a large amount of memory or have a high computation cost. To solve these problems, we pick TCAMs due to their extraordinary parallel comparison ability. Next, we explore some design choices and technical challenges on using TCAMs for the multi-match classification in Section 3.3. We present our Geometric Intersection Method to correctly order filters for reporting multi-match classification results in Section 3.4. Then we describe our method to efficiently represent these filters into TCAM in Section 3.5. We demonstrate the effectiveness of our TCAM-based method using the SNORT rule sets in Section 3.6. With our approach, we can report multi-match classification results with just two memory lookups while using a moderate TCAM size of 135 KB. Finally we briefly review the related work in Section 3.7 and conclude in Section 3.8.

3.1 Introduction

In typical packet classification, an incoming packet is compared against a set of filters. Most traditional applications, such as IP routing, only require the highest priority match, e.g., the longest prefix match. However, as we showed in Section 1.3, many new applications demand multi-match packet classification, where all matching filters need to be reported. For example, in accounting applications, multiple counters need to be updated for a given packet [55]. As different packets are associated with different sets of counters, multi-match classification is necessary to identify the relevant counters for each packet. Another application of multi-match classification is network intrusion detection systems, which monitor packets in a network and detect malicious intrusions or DOS attacks. Systems like SNORT [10] employ thousands of rules, each containing attack signatures. Each rule has two components: a *rule header* and a *rule option*. As we explained in Section 2.1.1, the rule header (the focus of this chapter) is a classification filter that consists of five fixed fields: protocol, source IP, source port, destination IP, and destination port. The rule option (the focus of Chapter 5 and Chapter 6) is more complicated: it specifies intrusion patterns to be used to scan packet contents. Rule headers may have overlaps, so a packet may match multiple rule headers. Multi-match classification is used to find all the rule headers that match a given packet so we can check the corresponding rule options one by one later.

The multi-match classification problem is relatively new, and we are among the first to study it in depth. The related single-match classification problem, however, has been extensively studied. We have surveyed in Section 2.2.1 various single-match classification algorithms. We showed that the state of art heuristic approaches [30, 32, 35, 36] provide the fastest classification results, with 20-30 memory accesses per packet in the "worst case".

Multi-match classification problem is more complex to implement than single-match classification since it needs all the matching results. Thus, some of the heuristic optimizations

used for the single-match classification do not apply for the multiple-match case, as we will show later in Section 3.2. Moreover, multi-match classification is usually the first step in performing complex network system functions followed by processing that is dependent on the classification results. For example, given a packet, it may match multiple counters and all these counters need to be incremented. Ramabhadran and Varghese showed that counter updating is a complicated process [56]. We first need to conduct a memory read to get the current value of the counter, then update the counter, and finally do a memory write to write the value back. As line rates have been increasing to 1 Gigabit or even 10 Gigabit rates (OC-192 is 10 Gbps), each counter matched by a packet must be read and written in the even decreasing time to receive a packet at line speeds. For example, a 40 byte packet must be processed in 32 nanoseconds (ns) at OC-192 speeds. A high number of updates impose stringent requirements on the bandwidth needs. For example, eight 64-bit counters in every 32 ns require 16 Gbps of memory bandwidth. These complicated follow-up processing demands multi-match classification to be finished in the order of several nanoseconds.

In this chapter, we adopt TCAMs for the multi-match classification problem due to their extraordinary parallel comparison ability. We present a TCAM-based approach, which produces multi-match classification results with only two memory lookups per packet. Before we present our algorithms in detail, next we study the characteristics of multi-match classification sets and understand their special requirements.

3.2 Characteristics of Multi-match Classification Sets

In this section, we compare some typical multi-match classification filter sets against single match classification filter sets. We show that multi-match filter sets are larger and create more intersections in Section 3.2.1. Due to these characteristics, we demonstrate that software-based approaches do not work efficiently on multi-match classification filter sets in

Table 3-1. SNORT rule headers statistics.

Version	Release date	Filter set size	Filters added	Filters deleted
2.0.0	4/14/2003	240	-	-
2.0.1	7/22/2003	255	21	6
2.1.0	12/18/2003	257	3	1
2.1.1	2/25/2004	263	6	0

Table 3-2. Comparison of multi-match and single-match filter sets.

	Multi-match filter sets (SNORT)				Single-match filter sets				
	2.0.0	2.0.1	2.1.0	2.1.1	S1	S2	S3	S4	S5
# of filters in the set	240	255	257	263	3061	184	4557	68	264
# of fields per filer	5	5	5	5	5	5	5	5	5
Average filter size (logarithm of 2)	64.67	60.73	60.81	60.91	31.6	54.86	24.5	56.58	48.25
# of intersections	3453	3754	3758	4067	3420	318	3502	70	318

Section 3.2.2. Finally, in Section 3.2.3, we explain the reasons that make TCAM a great candidate for the multi-match classification problem.

3.2.1 Study of Multi-match classification filter sets

We pick the SNORT rule sets [10] as the test sets for multi-match classification because they are very popular (downloaded by approximately 2000 users per week) and also they are publicly available. We tested all the publicly available Versions after 2.0. Although each set has around 1700-2000 rules, many of the rules share a common rule header (classification filter), because intruders piggyback on top of known vulnerabilities, so new attacks are built upon old ones. As illustrated in Table 3-1, unique rule headers in each version are relatively stable. Note that the versions that share the same rule headers with the previous version are omitted.

Table 3-2 compares the SNORT multi-match set with the single match filter sets used in real-world firewalls, core routers and access routers [46]. The first row records the number of

filters in the filter set and the second records the number of fields per filter. These filter sets all have the standard five fields: protocol, source IP, source port, destination IP, and destination port. The third row of the table records the average filter size. The size of a filter denotes the number of different packet headers it can match. For example, a very specific filter "*tcp 162.208.1.1 80 138.1.1.1* 80" matches only one specific tcp packet header that is from the source IP address *162.208.1.1* with a source port *80* to the destination IP address *138.1.1.1* with the destination port *80.* Therefore, this filter's size is one. At the other extreme, a very generic filter "*any any any any any*" will match packets with any protocol, any source, any port number, to any destination IP addresses with any port number. The unique packet headers it can match are exponential to the number of bits in the filter. In this case, the number of bits in the filter is 104 (8 bits protocol id, 2 ports with 16 bits each, 2 IP addresses with 32 bits each). Thus the size of this generic filter is 2^{104}. As shown in Table 3-2, multi-match filters are on average $2^{64.67}/2^{43.158} \approx 2^{20}$ times bigger than single matching filter sets.

Beside the filter size, we also record the number of intersections generated by the filter sets in Table 3-2. Here, two filters generate an intersection if they have overlaps; in other words, if there exists a packet that matches both filters. As we can see that, multi-match classification filters generate a much higher percentage of intersections than the single match classification filters. We show in the next section that large filter sizes and large numbers of intersections make software-based single-match solutions either need to consume a lot of memory or require many memory accesses.

3.2.2 Applying Single Match Classification Algorithms to the Multi-match Classification Set

The previous subsections showed that, compared to single-match filter sets, multi-match filter sets have two noticeable characteristics. First, filters are significantly larger in size. Second, they create a large number of intersections. In this section, we show that these two properties make some software-based heuristic algorithms, which are great for single match filter set, work poorly for the multi-match case.

Many schemes have been proposed for the single match classification problem. Some of them can be extended to report multi-match results. For example, Grid of Tries and Extended Grid of Tries (EGT) [32, 57] use the source and destination IP addresses to build a trie-like tree and store the related filters in the leaf nodes. By traversing the trie-like tree according the source and destination IP addresses, we can acquire all the related filters that apply to this source and destination IP combinations. Multi-match results can be obtained by comparing the other field information (e.g., protocol, source port, and destination port) with these filters one by one. Unlike EGT, where source and destination IP addresses are used to build the tree (trie in the EGT case), some state-of-the-art heuristic algorithms, such as HiCuts and Hypercuts [35, 36], use other criteria (such as port ranges) to build the tree. Similarly, they can be extended to provide multi-match classification results as well.

These software solutions were developed based on the characteristics of single-match filter sets. For example, EGT is based on the observation that in typical single-match filter sets, at most 20 rules will match any packet when considering the source and destination fields. Hence, we can search the tree based on the source and destination fields and then perform a linear search among the returned filters. This appears to be an economic solution. However, this observation is no longer valid for multi-match classification sets. As we showed in the previous section, for the SNORT filter sets, many filters intersect. For

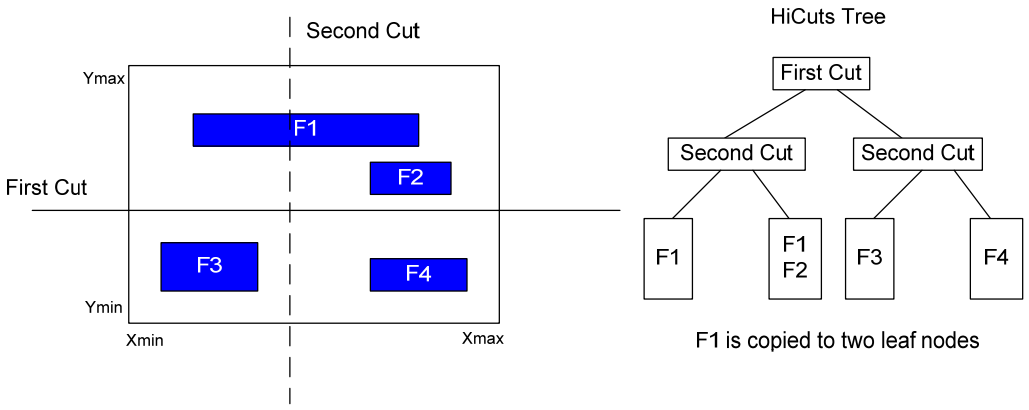


Figure 3-1. An example of the HiCuts algorithm.

Table 3-3. Applying HiCuts to the SNORT filter set.

SNORT Version	Tree height	Number of filters in leaf nodes	SRAM used(KB)
2.0.0	18	745,019	41,000
2.0.1	19	803,645	46,297
2.1.0	19	820,415	47,160
2.1.1	18	827,651	49,378

example, they frequently share some common source and destination addresses. Therefore, if we search the EGT-PC tree based on the source and destination addresses on the SNORT filter set, we found that EGT may return up to 153 filters for an input packet. All these filters have to be compared to the packet one by one in software, making the approach highly inefficient.

We also apply a popular tree based single-match classification scheme -- HiCuts to the SNORT rule sets. The HiCuts algorithm uses hyper planes to divide up the space and build the tree. Figure 3-1 illustrates an example, where there are four filters in a two dimensional space. The first cut goes horizontally and the second goes vertically. As we can see from the figure, Filter 1 is divided in two through the second cut, hence it is copied into two leaf nodes. When we apply the HiCuts algorithm to the multi-match filter sets, the large size of filters causes filters to be copied into many leaf nodes. Table 3-3 shows that, for the SNORT

2.0.0 filter set, there are 715019 filters in the leaf nodes. Since the SNORT filter set contains 240 unique filters, a filter is copied on average 745019/240 = 3108 times. This high degree of duplication results in a high demand for SRAM storage. More than 40 MBytes of SRAM are needed for the SNORT filter sets. This is beyond the largest single chip SRAM density (around 72Mbit = 9MB today [29]) and thus requires 6 SRAM chips. The number of SRAM interfaces per chip is usually limited. For example, the Intel IXP 2800 only has 2 SRAM interfaces [52]. In addition, a high degree of filter duplication calls for large update costs, as the insertion of a new filter or deletion of an old filter needs to touch all the leaf nodes that contain it.

As we can see from the above results, due to different characteristics of multi-match and single-match filter sets, existing software-based solutions do not work efficiently. They either require high computation costs or demand a large amount of memory. Instead, we need a hardware approach that uses a small amount of memory and requires few memory lookups to keep up with the high data rate. In the next section, we will explain why TCAMs meet these requirements.

3.2.3 Why TCAM?

The previous section showed that multi-match classification filter sets have unique characteristics that make them hard to solve using the single match software-based approaches. Furthermore, multi-match classification, because of the complex follow-up processing, is likely to have much tighter time requirements than the single match classification described in Section 3.1. We need an apparoach that uses a small amount of memory and requires few memory lookups to keep up with the high data rate. To meet these requirements, TCAMs are a good candidate because they are a kind of hardware built for parallel comparison. A TCAM compares the input with all its entries in parallel in hardware

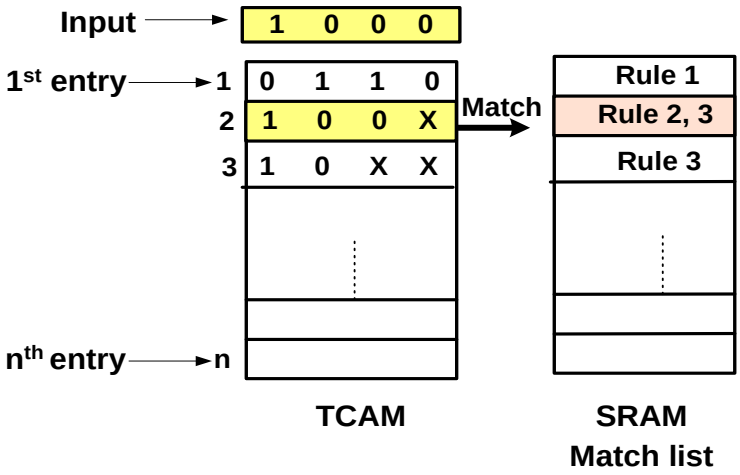


Figure 3-2. A TCAM with a SRAM match list.

and reports the matching result in a short time, e.g., 4 ns. They have three desired properties as we listed in Section 2.3.2: *high degree of parallelism, efficient support for IP addresses*, and *fast and deterministic lookup time*. Unlike software-based solutions, which are optimized for most common packets, TCAMs compare filters in hardware and report matching results in a deterministic time for all possible inputs. Hence, a TCAM-based solution offers more predictable performance and not vulnerable to certain malicious attacks.

These nice properties make TCAMs an excellent candidate for solving the classification problem at a multi-gigabit rate. As a result, over 6 million TCAM devices were deployed worldwide in 2004 [55, 58]. Given that TCAM is designed for single-match classification, in the next section, we will identity the technical challenges of using TCAMs for multi-match classification. Then in Section 3.4 and 3.5, we address these challenges and propose our algorithms for multi-match classification using TCAMs.

3.3 Technical Challenges

To use TCAMs for the multi-match classification problem, there are two challenges yet to be tackled: *filter ordering* and *negation representation*. In this section, we explain these in detail.

Challenge 1: Arrange filters in a TCAM compatible order

Currently, the commercially available TCAMs report only the first matching result if there are multiple matches. Figure 3-2 shows an example. This type of TCAM cannot directly report multi-match result. If one can change the TCAM hardware and let it return a bit vector of all matching results, one bit matching per matching entry, this still does not solve the multi-match problem. We still need to process the bit vector to extract the matching result pattern-by-pattern. Thus, the complexity is still O(N), in the number of patterns N. In the remainder of the chapter, we assume the off-the-shelf first-match TCAMs without any modifications.

Filters can have different relationships such as subset, intersection, and superset. These relationships can cause problems when reporting the matching results with first-match TCAMs. For example, suppose we have the following two filters:

(a) "Tcp $SQL_SERVER 1433 →$EXTERNAL_NET any"

(b) "Tcp Any Any → Any 139"

If we put filter (a) before (b) in the TACM, a packet matching both filters will report a match of (a) but never report (b), and vice versa. This is because filter (a) and (b) have an intersection relationship. That is, two filters have an interaction relationship if there exists a packet that matches both filters. In this case, we need an algorithm to add additional filters into the filter sets and order the filters in a specific way to avoid the above problem. Here, we define such an ordering a "*TCAM compatible order,*" which means: when a packet is compared with filters according to this storage order, we can retrieve all matching results solely based on the first matched filter. There should be no need to check the successive filters.

Challenge 2: Representing Negation with TCAMs

1xxx xxxx xxxx xxxx
x1xx xxxx xxxx xxxx
xx1x xxxx xxxx xxxx
xxx1 xxxx xxxx xxxx
xxxx 1xxx xxxx xxxx
xxxx x1xx xxxx xxxx
xxxx xx1x xxxx xxxx
xxxx xxx1 xxxx xxxx
xxxx xxxx 0xxx xxxx
xxxx xxxx x1xx xxxx
xxxx xxxx xx0x xxxx
xxxx xxxx xxx1 xxxx
xxxx xxxx xxxx 1xxx
xxxx xxxx xxxx x1xx
xxxx xxxx xxxx xx1x
xxxx xxxx xxxx xxx1

Figure 3-3. Binary representation of !80 in a TCAM.

The negation (!) operation is common in filter sets. For example, if we wish to find packets that are not destined for TCP port *80*, we will use a filter *"tcp any any → any !80"*. The 16-bit binary form of *80* is `0000 0000 0101 0000`. There is no direct way to map the negation into one TCAM entry. If we directly flip every bit, `1111 1111 1010 1111` stands for 65375, which is only a subset of *!80*. To represent the whole range of *!80*, we need 16 TCAM entries as shown in Figure 3-3. The basic approach flips one bit in one of the 16 binary positions and puts 'do not care' to all the others.

In addition to port negation, some filters require subnet addresses to be negated. For example, *$EXTERNAL_NET* frequently appears in filter sets, where $EXTERNAL_NET = !$HOME_NET. To represent this in the TCAM directly, we need to flip every bit in the prefix of $HOME_NET and put 'do not care' in the other positions. Because IP subnet addresses are 32 bits, each negated address costs up to 32 TCAM entries. Moreover, there could be several negations in one filter. For example, the filter "*tcp $EXTERNAL_NET any→ $EXTERNAL_NET !80"* requires up to a total of 32*32*16 = 16384 TCAM entries for this

single filter! This is obviously not an acceptable approach since TCAMs have a much smaller capacity than SRAMs (e.g., 1 MB with current technology).

The next two sections describe our approaches for addressing the above technical challenges. In Section 3.4, we develop an algorithm to automatically generate an extended filter set that is in a TCAM compatible order. Some of the filters in the extended filter set contain negation, which is not directly supported by TCAMs. To overcome this problem, in Section 3.5, we present an algorithm to remove these negations and map the extended filter set into the TCAM.

3.4 Geometric Intersection Method

This section addresses the first technical challenge as presented in Section 3.3, -- to order filters in a TCAM compatible order. To obtain multi-match results in one lookup with first-match TCAMs, we need to identify intersections between filters. Hence in this section, we first propose an overview of a *Geometric Intersection Method* for identifying filter intersections in Section 3.4.1. Next we study the relationships between filters in Section 3.4.2 and propose an algorithm to automatically order rules into a TCAM compatible order in Section 3.4.3.

3.4.1 Overview of the Geometric Intersection Method

In our Geometric Intersection Method, we maintain all the intersection filters as separate entries in the TCAM. Studies in [59, 60] show that the number of intersections between real-world filters is significantly smaller than the theoretical upper bound ($O(N^F)$, where N is the number of filters and F is the number of fields) because each field has a limited number of

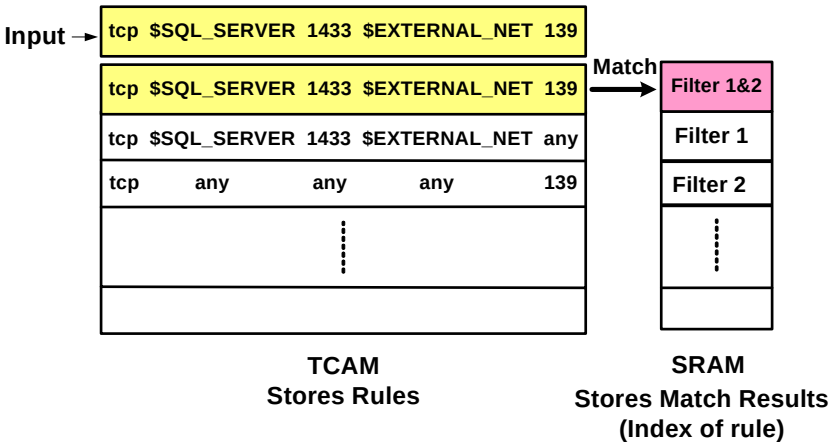


Figure 3-4. Geometric Intersection Method.

values (e.g., all known port numbers) instead of unconstrained random values. Hence, maintaining all the intersection filters is feasible.

Filters are inserted into the TCAM. Indices of the filters used to generate the intersection are stored in a list. We call this list a "Match List" and store it in SRAM as shown in Figure 3-4. Given a packet, we first perform a TCAM lookup and then use the matching index to retrieve all matching results with a secondary SRAM lookup. The extended filters we generate plus the original filters form an *extended filter set*. Throughout the remainder of this chapter, a "filter" refers to a member of the extended filter set, unless otherwise specified as a member of the original filter set. The items in the match list are the indices of filters in the original filter set.

Rules in the TCAM must follow a TCAM compatible order. Next, we study the relationship between filters and present an algorithm to automatically generate an extended filter set in this order.

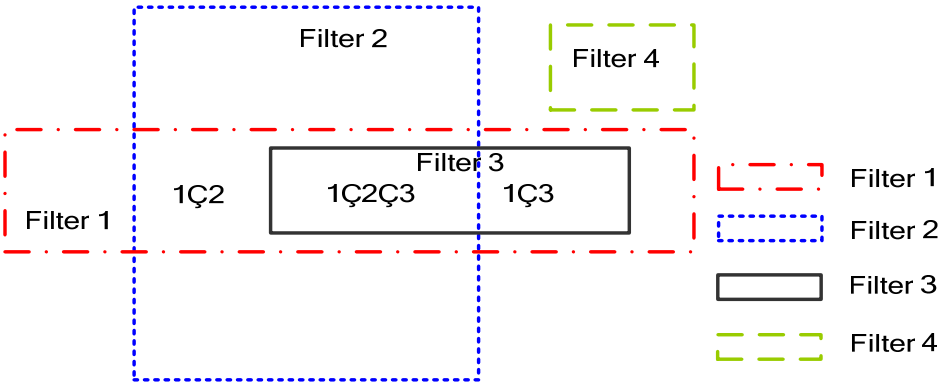


Figure 3-5 An example of intersections of three filters.

3.4.2 Relationship between Filters

Before we describe our algorithm for generating extended filter sets in a TCAM compatible order, let us first study the relationship between filters. There are four possible cases between any two different filters F_i and F_j,: exclusive, subset, superset, and intersection.

Case 1: Exclusive ($F_i \cap F_j = \phi$), e.g., Filter 2 and Filter 4 in Figure 3-5.

Case 2: Subset ($F_i \subseteq F_j$), e.g., Filter 3 is a subset of Filter 1 in Figure 3-5.

Case 3: Superset ($F_j \subset F_i$), e.g., Filter 1 is a superset of Filter 3 in Figure 3-5.

Case 4: Intersection ($F_i \cap F_j \neq \phi$), e.g., Filter 1 and Filter 2 intersect with each other in Figure 3-5.

As defined in Section 3.3, a TCAM compatible order requires filters to be ordered so that the first match should record all the matching results in the match list. Mapping this requirement into these four cases, two different filters F_i and F_j with match list M_i and M_j, we get the following requirements regarding the order of F_i and F_j in the extended rule set:

Requirement 1: Exclusive ($F_i \cap F_j = \phi$): then i and j can have any order.

Requirement 2: Subset ($F_i \subseteq F_j$): then $i<j$ and $M_j \subseteq M_i$.

Requirement 3: Superset ($F_j \subset F_i$): then $j<i$ and $M_i \subseteq M_j$.

Requirement 4: Intersection ($F_i \cap F_j \neq \phi$): then there is a filter $F_l = (F_i \cap F_j)$ ($l<i$, $l<j$), and $(M_i \cup M_j) \subseteq M_l$.

Requirement 1 is trivial: if F_i and F_j are disjoint, they can be in any order since every packet matching F_i never matches F_j. For Requirement 2 where F_i is a subset of F_j, every packet matching F_i will match F_j as well, so F_i should be put before F_j and match list M_i should include M_j. In this way, packets first matching F_i will not miss matching F_j. Similar operations are required for Case 3. Besides these three cases, partially overlapping filters lead to Case 4. In this case, we need a new filter F_l recording the intersection of these two filters ($F_i \cap F_j$) placed before both F_i and F_j with both match results included in its match list ($M_i \cup M_j \subseteq M_l$). Note that the intersection of F_i and F_j may be further divided into smaller regions by other filters (e.g., the intersection of Filter 1 and Filter 2 intersects with Filter 3 in Figure 3-5). In this case, all the smaller regions ($1 \cap 2$ and $1 \cap 2 \cap 3$) have to be presented before both F_i and F_j (Filter 1 and Filter 2 in our example). This can in fact be deduced by Requirement 4.

3.4.3 Generating a TCAM Compatible Order

We have studied all the possible relationships between any two filters. Along with the study, we have identified the ordering requirements of two arbitrary filters as given in Requirements 1 through 4. By applying the corresponding operations, we can meet the requirements and get a TCAM compatible order. Figure 3-6 is the pseudo-code for creating such an order.

The algorithm takes the original filter set $R = \{R_1, R_2, ..., R_n\}$ as the input. Each filter R_i is associated with a match list, which is an index of itself ($\{i\}$). The algorithm outputs an extended filter set F in the TCAM compatible order. The algorithm inserts one filter at a time into the extended filter set F, which is initially empty (the empty set obviously follows the requirements of the TCAM compatible order). Next, we show that after each insertion, F still

```
Extend_filter_set(R){
        F = φ ;
        For all the filter R_i in R
                F = Insert(R_i, F);
        return F;
}
Insert(x, F){
    for all the filter F_i in F {
        Switch the relationship between F_i and x:
            Case exclusive:
                continue;
            Case subset:
                M_i = M_x ∪ M_i;
                        continue;
            Case superset:
                M_x = M_x ∪ M_i;
                   add x before F_i ;
                return F;
            Case intersection:
                If (F_i ∩ x ∉ F and M_x ⊄ M_i)
                        add t = F_i ∩ x before F_i ;
                        M_t = M_x ∪ Mi
        }
       add x at the end of F and  return F;
}
```

Figure 3-6. Code for generating a TCAM compatible order.

meets the requirements. *Insert(x, F)* is the routine to insert filter x into F. It scans every filter F_i in F and checks the relationship between F_i and x. If they are exclusive, then we can bypass F_i. If F_i is a subset of x, we just add match list M_x to M_i and proceed to the next filter. If F_i is a superset of x, we add x before F_i according to the Requirement 3 and ignore all the filters after F_i (please refer to Appendix A for proof). Otherwise, if they intersect, then according to the Requirement 4, a new filter $F_i \cap x$ is inserted before F_i if it is not already been added. The match list for the new filter is $M_x \cup M_i$. We strictly follow the four requirements when adding every new filter, so the generated extended filter set F is in the correct TCAM compatible order.

Table 3-4. Example of original filter set with 3 filters.

	Original filter sets
1	Tcp $SQL_SERVER 1433 → $EXTERNAL_NET any
2	Tcp $EXTERNAL_NET 119 → $HOME_NET any
3	Tcp any any → any 139

Table 3-5. Extended filter set in a TCAM compatible order.

Extended Filters	Match List
Tcp $SQL_SERVER 1443 →$EXTERNAL_NET 139	1,3
Tcp $SQL_SERVER 1433 →$EXTERNAL_NET any	1
Tcp $EXTERNAL_NET 119 →$HOME_NET 139	2,3
Tcp $EXTERNAL_NET 119 → $HOME_NET any	2
Tcp any any → any 139	3

Table 3-4 is an example that illustrates the algorithm. It contains three filters. To generate extended filter set *F,* first we insert Filter 1. Filter 2 does not intersect with Filter 1 so it can be added directly. Now, we have Filter 1 followed by Filter 2. When inserting Filter 3, we find that it intersects with both Filter 1 and Filter 2, so we add two intersection filters with match list {1, 3} and {2, 3} and put Filter 3 at the bottom of the TCAM. The final extended filter set *F* is presented in Table 3-5. Notice that the first four filters contain negation ($EXTERNAL_NET, which is the negation of HOME_NET), as we have explained in Section 3.3, filters containing negation consume a large amount of TCAM entries. In the next section, we present a scheme to remove the negations in the filter sets so as to map filters efficiently into the TCAM.

3.5 Negation Removing

The scheme we just presented can be used to generate a set of filters in a TCAM compatible order. The next job is to efficiently insert these filters into the TCAM. As explained in Section 2.3.2, each cell in the TCAM can take one of three states: 0, 1 or ‘do not care’. Hence, each filter needs to be mapped into a representation composed of these three states.

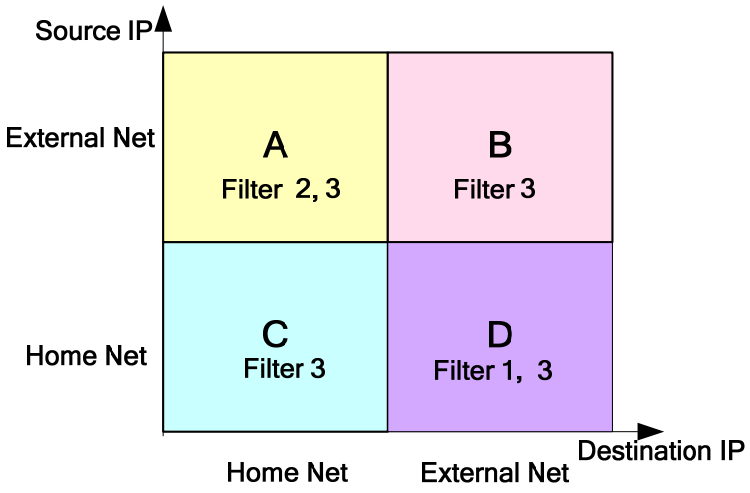


Figure 3-7. Source and destination IP addresses space.

Usually, a filter contains IP addresses, port information, protocol type, etc. IP addresses in the CIDR form can be represented in the TCAM using the 'do not care' state. However, the port number may be selected from an arbitrary range. Liu [61] has proposed a scheme to efficiently solve port range problem. However, we do not use this scheme here because it requires two additional memory lookups. Furthermore, the SNORT filter set does not contain a huge number of ranges. Instead, we just directly map the range into the TCAM using multiple entries, e.g., port 2-5 is represented as 01* and 10*. A more complicated problem for the TCAM is that some IP and port information is in a negation form. As explained in Section 3.3, each negation consumes many TCAM entries, so in this section, our goal is to remove negation from the filter set to save TCAM space.

Before presenting our scheme, let us first look at the combinations of source and destination IP address spaces as shown in Figure 3-7. Use the filter set in Table 3-4 as an example, Filter 3 applies to all 4 regions since it is "any" source to "any" destination; Filter 1 applies to Region D because we assume $SQL_SERVER is in side $HOME_NET; and Filter 2 applies to Region A.

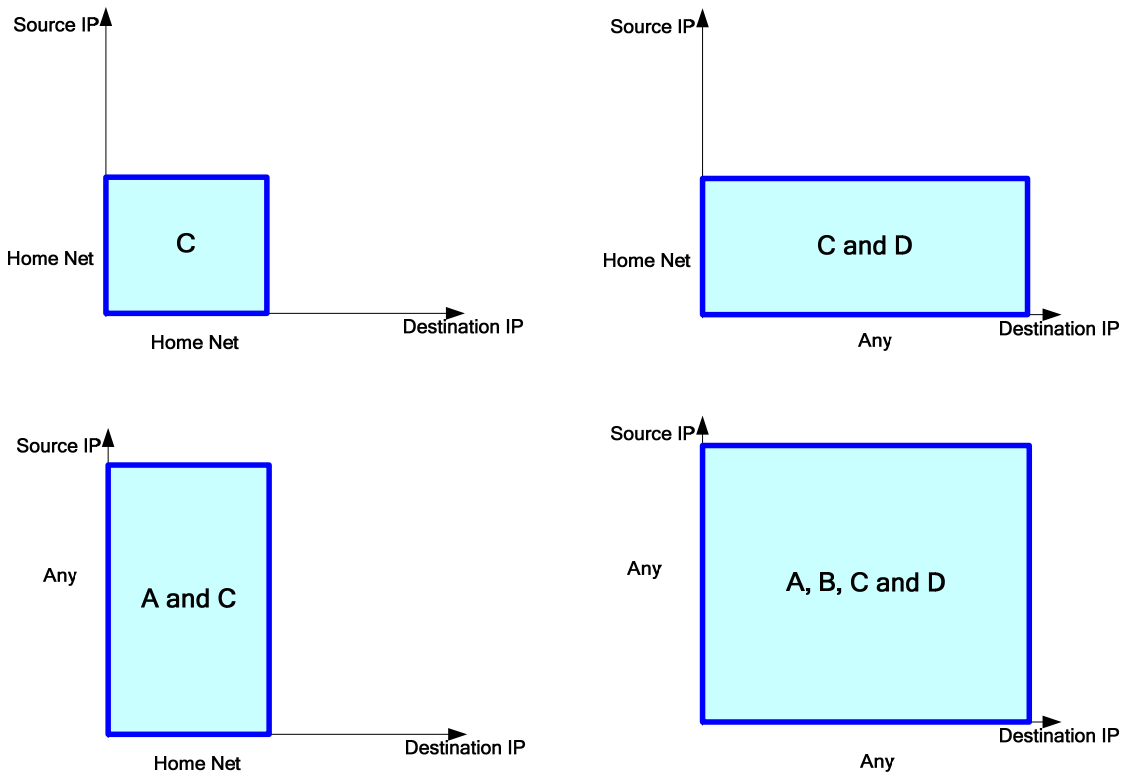


Figure 3-8. Regions that do not contain negation.

In this two dimensional space, there are regions that do not contain negations as shown in Figure 3-8. They are Region C ($HOME_NET to $HOME_NET), the combination of Region C and Region D ($HOME_NET to any), the combination of Region A and Region C (any to $HOME_NET), and the whole space (any to any). On the other hand, there are regions that contain negation ($EXTERNAL_NET), including Region A ($EXTERNAL_NET to $HOME_NET), D ($HOME_NET to $EXTERNAL_NET), and B ($EXTERNAL_NET to $EXTERNAL_NET) as shown in Figure 3-9.

Consider a region containing negation, Region A, as an example: the filters in this region are in the form of *"* $EXTERNAL_NET * →$HOME_NET^{+} *"*. Note that * means it could match any thing (e.g. "tcp" or "any" or a specific value). $HOME_NET^{+} stands for $HOME_NET and any subset of it such as $SQL_SERVER. If we can extend filters in Region A to Region A and C as shown in Figure 3-10, we can replace $EXTERNAL_NET

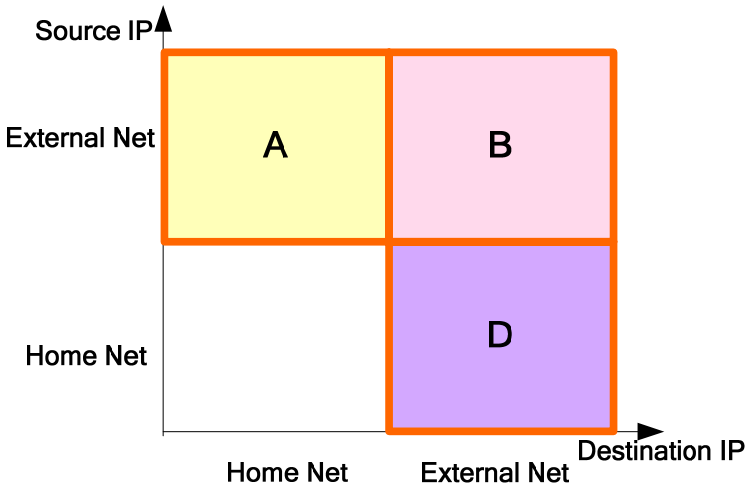
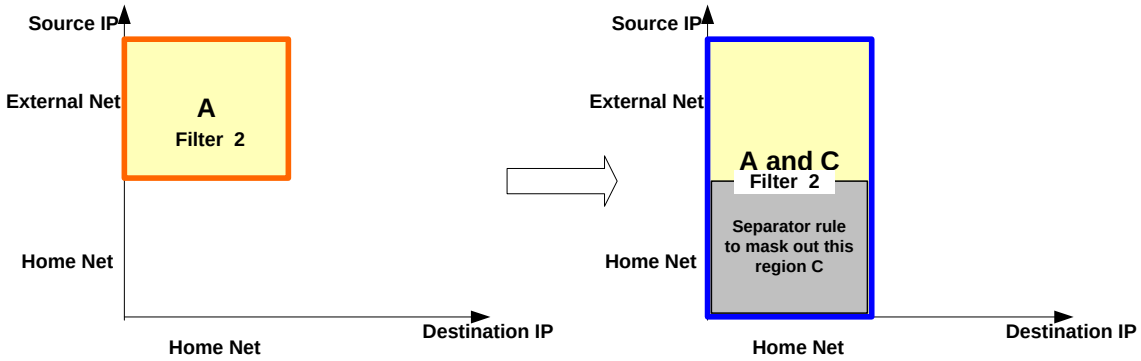


Figure 3-9. Regions that contain negation.



Only one rule in TCAM with negation
1. Filter 2

Two rules in the TCAM both without negations
1. A separator filter captures all packets in Region C
2. Filter 2 is extended to region A and C

Figure 3-10. An illustration of our negation removing scheme.

with keyword "any" and now filters are in the format of *"* any * →$HOME_NET⁺ *"*. However, after extending the region, we change the semantics of the filter and this may affect packets in Region C. In other words, packet *"* $HOME_NET * → $HOME_NET⁺ *"* will report a match of this filter as well.

This problem, however, is solvable because TCAMs only report the first matching result. With this property, we can first extract all the filters applying to Region C and put those at the top of the TCAM. Next, we add a separator filter between Region C and Region A*: "any*

$HOME_NET any → $HOME_NET any" with an empty action list. In this way, all the packets in Region C will be matched first and thus ignore all the filters afterwards. With this separator filter, we can now extend all the filters in Region A to Region A and C. Similarly, filters in Region D can be extended to Region C and D, filters in Region B can be extended to Region A, B, C, D. Therefore, we will put all the filters in the following order:

Filters in Region C: "* $HOME_NET+ * →$HOME_NET+ *"
Separator 1: "any $HOME_NET any →$HOME_NET any"
Filters in Region D, specified in the form of Region C and D:
"* $HOME_NET+ * →any *"
Filters in Region A, specified in the form of Region A and C:
"* any * →$HOME_NET+ *"
Separator 2: "any $HOME_NET any → any any"
Separator 3: "any any any →$HOME_NET any"
Filters applying to Region B, specified in the form of Region A, B, C and D:
"* any * →any *"

Putting extended filter sets in this order can be achieved simply by first adding all three separator filters to the beginning of the original filter set, then following the algorithm in Section 3.3. If a filter applies to Region A, it will automatically intersect with the separator 1, and generate a new filter in Region C. If a filter applies in Region B, it will intersect with all three separators and create three intersection filters. After that, we can replace all the $EXTERNAL_NET references with the keyword "any". Table 3-6 shows the result of mapping the filter set of Table 3-4 into the TCAM. The first filter in Region C is extracted from Filter 3 that applies to all four regions. The second filter is a separator filter. With these two filters, we can replace the $EXTERNAL_NET in filters 3-6 with the keyword "any". At the end, there is filter 7 that applies to all the regions. Separator filters 2 and 3 are omitted because no filter is in the form of $EXTERNAL_NET to $EXTERNAL_NET in the original filter set. In this example, by adding only two filters, we can completely remove the

Table 3-6. Extended filter set in a TCAM with no negation.

TCAM Index	TCAM entries	Match list
1	Tcp $HOME_NET any →$HOME_NET 139	3
2	any $HOME_NET any →$HOME_NET any	
3	Tcp $SQL_SERVER 1433 → any 139	3, 1
4	Tcp $SQL_SERVER 1433 → any any	1
5	Tcp any 119 → $HOME_NET 139	2,3
6	Tcp any 119 →$HOME_NET any	2
7	Tcp any any →any 139	3

$EXTERNAL_NET. Compared this to the approach in Table 3-5, which needs up to 4*32 +1 = 129 TCAM entries, this results in drastic 94.5% reduction in space usage.

The above example is a special case because there is only one type of negation ($EXTERNAL_NET) in one field. In the more general case, there can be more than one negation in each field. For example, there could be both !80 and !90 or !subnet1 and !subnet2 in the same field. The proposed scheme can be easily extended to handle this. If there are *k* unique negations in one field and their non-negation forms do not intersect (e.g., an exact match of 80 and an exact match of 90), then we need *k* separators of the non-negation form (80, 90), which may be in any order. If they intersect, we need up to $2^k - 1$ separation filters for this field. For instance, suppose the negations are !subnet1 and !subnet2. There should be three separation filters applying to {subnet1 ∩ subnet2}, subnet2, and subnet1. *k* is usually a very small number because it is limited by the number of peered subnets. In general, if each field *i* needs k_i separators, then at most $(\prod(k_i+1))-1$ separator filters should be added. In our previous example of removing $EXTERNAL_NET from source and destination IP addresses, $k_1 = k_2 = 1$, so we need a total of 2*2-1 = 3 separator filters.

We presented our Geometric Intersection Method for generating an extended filter set in a TCAM compatible order in Section 3.2. In this section, we proposed an algorithm to efficiently remove the negations in the extended filter set so as to efficiently map the filters into the TCAM, saving the TCAM space. With these schemes, we can obtain multi-match

classification results with just two memory lookups as showed in Figure 3-4: one TCAM lookup to get an index into SRAM, and then an SRAM lookup using the index to retrieve all the matching results. Next, we test our algorithms using different multi-match filter sets.

3.6 Simulation Results

In this section, we test the effectiveness of our TCAM-based approach. We first show that we can use our Geometric Intersection Method to generate multi-match classification results in Section 3.6.1. Then we show the negation removing scheme can efficiently map filters into the TCAM in Section 3.6.2. To test the effectiveness of our algorithm, we use the SNORT [10] filter set, which was introduced in Section 3.2.1. We tested all the publicly available versions after 2.0 as shown in Table 3-1.

3.6.1 Effect of Geometric Intersection Method

Our goal is to put multi-match classification filters (SNORT rule headers) into a TCAM as classification filters, and store the corresponding matching filter indices in the match list. Hence, given an incoming packet, with one TCAM lookup and another SRAM lookup, we can get the multi-match packet classification result.

We use the Geometric Intersection Method to produce extended filter sets that are in TCAM compatible orders. The second column in Table 3-7 records the sizes of the extended filter sets. The number of intersections SNORT filter sets generated is significantly lower than the theoretical upper bound. It is roughly 15 times the original filter set, which is well below the theoretical upper bound. This agrees with the findings in [30, 36, 59]. Hence the Geometric Intersection Method is viable for the SNORT filter set.

Table 3-7. Statistics of extended filters set.

Version	# of filters in extended set	Single negation	Double negations	Triple negations
2.0.0	3,693	62.334%	0.975%	0
2.0.1	4,009	62.484%	1.422%	0.025%
2.1.0	4,015	62.540%	1.420%	0.025%
2.1.1	4,330	62.332%	1.363%	0.023%

Table 3-8. Performance of negation removing scheme.

SNORT Version	With Negation		Negation Removed		TCAM space saved
	Extended filter set size	TCAM entries needed	Extended filter set size	TCAM entries needed	
2.0.0	3,693	120,409	4,101	7,853	93.4%
2.0.1	4,009	145,208	4,411	8,124	94.4%
2.1.0	4,015	145,352	4,420	8,133	94.4%
2.1.1	4,330	151,923	4,797	8,649	94.3%

3.6.2 Negation Removing Scheme

The number of negations in the extended filter set is significant. As shown in Table 3-7, on average 62.4% of the filters have one negation, 1.295% of the filters have two negations and there are a very small number of filters with three negations. In our simulation, we assume the home network is a Class C address with a 24 bit prefix, so each $EXTERNAL_NET needs 24 TCAM entries. Negation of a port, e.g., !80, !21:23 consumes 16 TCAM entries. Under this setting, a single negation takes up to 24 TCAM entries; a double negation consumes up to 24*24 = 576 TCAM entries; and a triple negation requires up to 24*24*16 = 9216 TCAM entries. Hence, if directly putting all the filters with negation into the TCAM, it takes up to 151,923 TCAM entries as shown in the third column of Table 3-8. Our negation removing scheme presented in Section 3.4 significantly saves TCAM space. For the SNORT filter header set, we added 2*3*2*2-1 = 23 separation filters in front of the original filter set because there are four types of negations: $EXTERNAL_NET at source IP, $EXTERNAL_NET at destination IP, !21:23 and !80 at source port, and !80 at destination

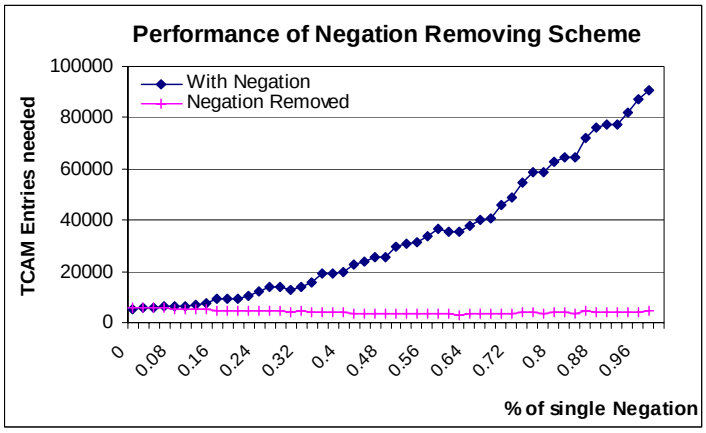


Figure 3-11. Negation removing scheme.

port. This approach only adds about 10% extra filters in the extended filter set (4th column of Table 3-8). However, with these additional filters, the number of TCAM entries is reduced by over 93%.

Note that this total number is larger than the extended filter set size. This is because some filters contain port ranges that consume extra TCAM entries. The range mapping approach in [61] is not used because this approach requires two additional memory lookups for key translations, which reduces classification speed. If a lower speed is acceptable, then we can also incorporate the range mapping technique. In this case, the total TCAM entries needed is just the size of extended filter set after removing negations.

Each filter is 104 bits (8 bits protocol id, 2 ports with 16 bits each, 2 IP addresses with 32 bits each), which can be rounded up to use a TCAM with 128 bits per entry. The total TCAM space needed for the SNORT filter header set in such a TCAM is only 128*8649 = 135 KB.

To study the effect of negation, we randomly vary the negation percentages in the original filter set. In the SNORT original filter header sets, 89.7% of the filters contain single negation and 1.1% contain double negation. So, we first test the single negation case. Figure 3-11 shows the TCAM space needed both with and without our scheme for negation removal. When the percentage of negation is very low, the two schemes perform similarly. In fact,

when the negation percentage is very small (<2%), putting negation directly using the straight forward solution shown in Figure 3-3 is better than the proposed scheme since we introduce extra separation filters that may intersect with other filters. However, as the percentage of negation is higher, the TCAM space needed for the "with negation" case grows very fast. In contrast, the curve of the proposed scheme remains flat and thus can save a significant amount of TCAM space. For example, when 98% of the filters involve negation, the proposed scheme saves 95.2% of the TCAM space as compared to the "with negation" case. This is only for the single negation case. For double negations, or triple negations, the savings would be even greater since each double/triple negation filter requires many more TCAM entries.

3.7 Related Work

Most existing software-based packet classification algorithms are designed specifically for single-match classification. The most relevant work on multi-match classification is on filter conflict detection in [62]. Here, conflicts denote a packet matching multiple filters. The single-match classification only allows one final matching result and some of the matching results conflicts with each other, e.g., instructions to "drop the packet" or "accept the packet". To resolve these conflicts, usually filters are sorted in a particular order. However, [62] shows that there are cases where this commonly used conflict resolution scheme does not work. For these cases, they proposed to solve the problem by adding new filters in a manner similar to our approach. However, they emphasize on finding these conflicts, while we aim to automatically generate intersection rules and efficiently map them into the TCAM, saving TCAM space.

There have been extensive studies of the single-match classification problem. The single-match problem on multiple fields is complex [59]. For *N* arbitrary non-overlapping regions in

F dimensions, it is possible to achieve an optimized query time of $O(\log(N))$, with a complexity of $O(N^F)$ storage in the theoretical worst case [29, 35]. However, real-world filter sets are typically simpler than the theoretical worst case, and heuristic approaches [32, 35, 63] provide faster results, e.g., 20-30 memory accesses per packet in the "worst case". These software approaches were developed based on typical characteristics of single match filter sets. We showed in Section 3.2 that two representative algorithms (EGT-PC and HiCut) either compare the input packet against many filters one by one, or require a large amount of memory when applied to the SNORT filter sets.

Most recently, TCAMs are used in high end routers for single-match packet classification for routing and Quality of Service (QoS) services [61, 64, 65]. Since TCAMs are smaller and more expensive than SRAMs, different approaches are proposed to save TCAM space or reduce TCAM power consumption. For example, Liu [12] proposed an algorithm for mapping range values into TCAMs. CoolCAMs [14] partitioned a TCAM so that for a given packet, it searches only several partitions to achieve lower power consumption. Spitznagel, et al. [66], extended this idea and organized the TCAM as a two level hierarchy in which an index block is used to enable/disable the querying of the main blocks. In addition, they also incorporated circuits for range comparisons within the TCAM memory array. These works focused on single-match classification whereas this chapter aims to find a TCAM-based multi-match solution.

Shortly after the publication of our work, Lakshminarayanan, et al., proposed another TCAM-based approach to the multi-match packet classification problem [55]. We will explain their approach in detail later in Section 4.2.3 and compare them to our scheme in Section 4.4

3.8 Conclusions

In this chapter, we presented a TCAM-based method to solve the multi-match classification problem. Through inserting interactions filters into the TCAM, our scheme reports all the matching results with a single TCAM lookup and a SRAM lookup. In addition, we propose a scheme to remove negation in the filter sets, thus saving 93% to 95% of the TCAM space over a straightforward implementation. From our simulation results, the SNORT filter header set can easily fit into a small TCAM of size 135 KB, and is able to retrieve all matching results within just one TCAM access and one SRAM access.

Our Geometric Intersection multi-match method inserts interaction filters into TCAMs to report multi-match classification results. Theoretically, the number of intersection filters can be $O(N^F)$, where N is the total number of filters, and F is the number of fields. Real-world filter sets are typically simpler than the theoretical worst case, but we still observe that the SNORT rule header set creates more than ten times more intersections than the original filter set size. When the filter set generates many intersections, the method based on Geometric Intersection is expensive both in memory size and thus power consumption. We address these issues in the next chapter.

4 Energy Efficient Multi-match Packet Classification

In the previous chapter, we adopt TCAMs for solving the multi-match classification problem due to their fast parallel matching capability. Our Geometric Intersection Method reports the multi-match classification results with just one TCAM lookup and one SRAM lookup. It achieves this high classification speed by adding intersection filters into the TCAM. Although processing speed is high, if there are a large number of intersections in the input filters, there can be a significant increase in memory and power consumption. Our experimental results on the SNORT filter sets show that this scheme increases the number of filters in the TCAM by a factor of 10, hence consuming a large amount of TCAM memory. In addition, these intersection filters are all compared in parallel in the TCAM, resulting in a high power usage because the TCAM energy consumption is linear to the number of entries searched in parallel. However, for applications that set highest priority on classification speed and have sufficient memory, our scheme processes packets at a very high rate.

In this chapter, we develop a general and balanced scheme that takes into consideration classification speed, memory consumption and power consumption. We present a new Set Splitting Algorithm (SSA), which splits filters into multiple groups while performing separate TCAM lookups within these groups. It guarantees the removal of at least half of the intersections when a filter set is split in two, thus reducing at least 50% of the TCAM memory and power consumption caused by intersections. SSA also accesses filters in the TCAM only once per packet, further reducing power consumption. Our simulation results based on the SNORT filter sets show that, compared with the Geometric Intersection Method proposed in the previous chapter, SSA uses 90% less TCAM memory and power at the cost of one more TCAM lookup per packet.

4.1 Introduction

Multi-match classification is among the initial steps in choosing a set of functions to be performed when processing a packet. For example, only after obtaining the multi-match classification results may the router increment follow-up counters, such as those for protocol type and source IP addresses. These actions are performed on every packet, so we do not want the multi-match classification process to be the bottleneck in the system. We showed in Section 3.1 that a purely software-based approach cannot operate at speeds that are consistent with multi-gigabit line rates. To maintain high packet processing rates, we adopt TCAMs for multi-match classification, as they perform fast parallel searches across all filters in hardware. We proposed a Geometric Intersection-based approach in the previous chapter. Shortly the publish of our work, Lakshminarayanan, et al., proposed another scheme called MUD for multi-match classification also using TCAMs [55]

However, TCAMs have limitations as we stated in Section 2.3.2. They cost about 30 times more per bit of storage than DDR SRAMs and consume 150 times more power per bit than SRAMs. As a result, in some high end routers, TCAMs consume around 30 to 40 percent of the total line card power. As line cards are stacked together, TCAMs impose a high cost on the cooling system. The energy consumption level of a TCAM is related to the TCAM size and the access frequency [65]. Figure 4-1 illustrates the relationship between TCAM memory consumption, the number of TCAM accesses, and the TCAM power consumption. The energy used by a TCAM grows linearly with the number of entries searched in parallel and also scales with the frequency of TCAM accesses. For example, if we double the TCAM memory, the energy consumption also doubles. Similarly, if we access the TCAM twice as frequently, it uses two times as much power. If memory and access frequency are doubled simultaneously, the power consumption grows to four times the

original. To be cost and energy efficient, TCAM-based multi-match solutions must *use an economical TCAM memory size* and *perform a limited number of TCAM lookups for each packet.*

None of the previously published multi-match classification schemes can satisfy both requirements. For example, the MUD scheme proposed by Lakshminarayanan, et al., encodes the extra bits in each TCAM entry to support range and multi-match lookup [55]. The amount of TCAM memory needed is linear with the size of the filter sets. However, MUD needs at least k TCAM lookups to get k matching results, and all the entries in the TCAM are accessed during each lookup. We show in Section 4.4.2, k can be 12 - 20 for the SNORT filter sets, resulting in a proportional 12-20 times increase in power consumption or processing delay compared to the Geometric Intersection-based method. The effect is much longer processing time and higher power consumption for packets that match many filters as shown on the upper right side of Figure 4-2. Our Geometric Intersection Method proposed in Chapter 3 reports classification results in one TCAM lookup and one SRAM lookup [67 2004]. However, achieving this speed requires that all filter intersections (regions of overlap between filters) be inserted as new filters in the TCAM. Theoretically, N filters with F fields can create $O(N^F)$ intersections, thereby adding $O(N^F)$ entries to the TCAM. This approach is not cost or energy efficient when filters have many intersections as shown in the upper left side of Figure 4-2.

The Set Splitting Algorithm (SSA) is both memory and power efficient. It works by splitting the filters into several sets and performing separate TCAM lookups for each. Through this filter splitting technique, we can decrease the number of intersections required to be stored in the TCAM, thus decreasing the TCAM memory consumption and also its power consumption as shown in Figure 4-2. In addition, SSA performs a deterministic number (2-4 for the SNORT rule set) of TCAM lookup per packet, yielding high

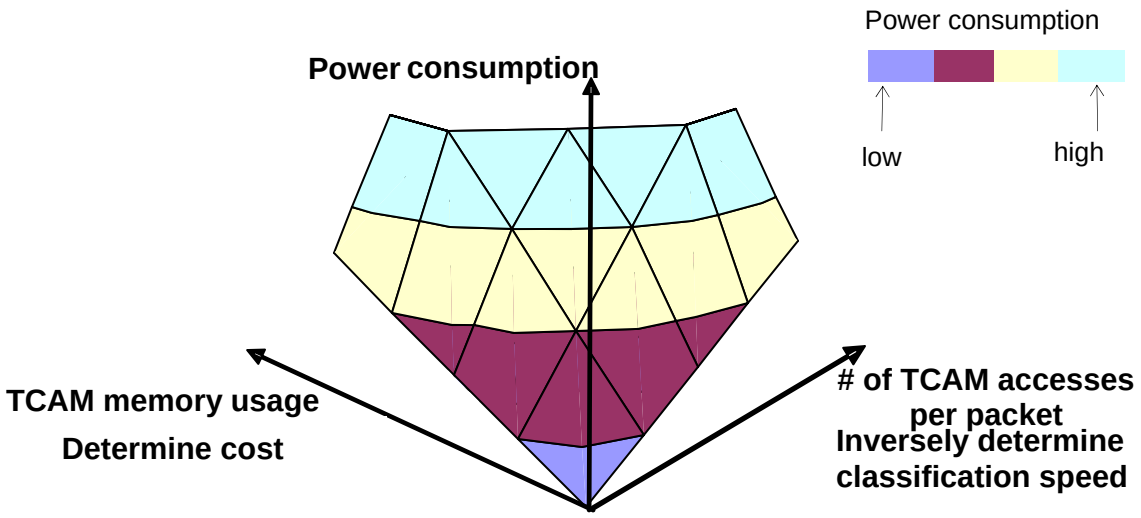


Figure 4-1. Relationships between TCAM memory consumption, number of TCAM accesses, and TCAM power consumption.

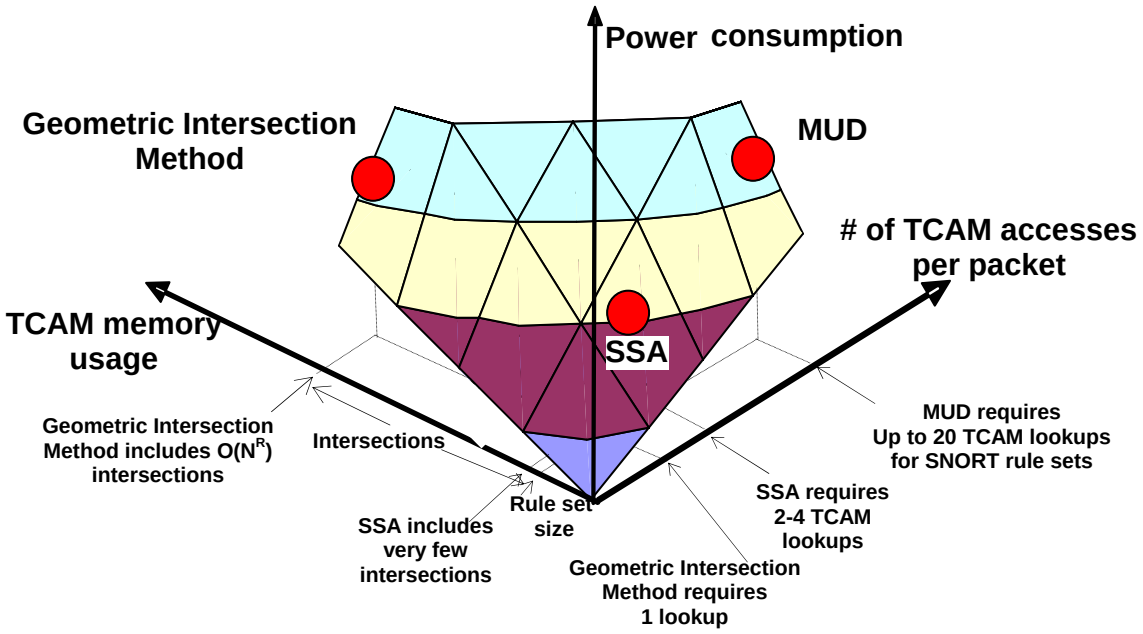


Figure 4-2. Power consumption of different TCAM-based approaches.

classification rates. The rest of the chapter is organized as follows. We start by reviewing existing TCAM-based approaches for the multi-match classification problem in Section 4.2 and show that none of them is both memory and power efficient. Then we present our SSA scheme in Section 4.3. We demonstrate the effectiveness of our approach by comparing it with two previously published TCAM-based schemes (MUD and the Geometric Intersection Method) in Section 4.4. Simulations on the SNORT filter sets show that SSA uses approximately the same amount of TCAM memory as MUD, but yields a 75% to 95% reduction in power consumption and up to 4 times speedup. Compared with the Geometric Intersection Method, SSA uses 90% less TCAM memory and power at the cost of one additional TCAM lookup per packet.

4.2 Related Work

Several approaches have been proposed to save TCAM space and reduce TCAM power consumption. For example, methods like CoolCAMs [47] and load balancing TCAMs [68, 69] modify the TCAM and partition it so that, for a given packet, only a limited number of partitions are searched to decrease power consumption. These approaches are designed for one dimensional packet classification, such as destination IP lookup (also called longest-prefix match lookup), which is a common network router operation. Spitznagel, et al., [66] proposed a extension for the multi-dimensional case. They modify the commercially available TCAM so as to reorganize it into a two level hierarchy where an index block is used to enable/disable the access of the main blocks in TCAM (please refer to Section 2.3.2 for the description of block). They also incorporated circuits for range comparisons. In addition, these approaches are all designed for single match packet classifications and thus report only the highest priority match. In the rest of this section, we review current TCAM-

based approaches to the multi-match classification problem and point out when they fall short in providing a complete solution.

4.2.1 Bit Vector Approach

Currently, commercial TCAMs only report one matching result (usually the first match). This is because TCAMs have priority encoding circuits that take the matching vector and output the first matching index, as previously discussed in Figure 2-5 in Section 2.3.2. If we can modify the TCAM and remove that priority encoder, it can output a bit vector of matching results, one bit for each entry. This works very efficiently when each matching result is connected directly to a hardware processing unit [70]. The related follow-up processing can be triggered immediately, and these follow-up processing units can run in parallel.

However, if we don't have the whole system built with the aforementioned hardware, the bit vector approach involves significant processing overhead. In packet classification architectures, a processor (CPU or Network Processing Unit (NPU)) is connected to a TCAM. The processor sends packet information to the TCAM, and the TCAM sends back the matching results. Using the matching results, the processor performs the relevant operations on the packet (e.g., send to a port, update a counter). If the TCAM returns a matching vector, the processor needs to step through the vector to extract the matching results. This is not efficient when the number of entries in the TCAM N is large and the matching vector is sparse, which is true for SNORT filter sets as we will show later in Section 4.4. The inefficiency comes from two reasons. First, the rate to transfer the N bit vector is limited by memory bandwidth. Second, processing complexity is $O(N)$ to extract the matching results.

It is hard to change the priority encoder to output the matching results only, because the number of matching entries and how they are spread over the bit vector vary for different

applications. It is also difficult for TCAM vendors to come up with a general design that works for all applications.

4.2.2 Current Industrial Approaches

Some commercial TCAMs support multiple matching. There is a valid bit for each TCAM entry that indicates whether or not to compare it with the input as we shown in Section 2.3.2. These entries are initially set to valid. Given an input, the first match will be reported in the first cycle. The valid bit for the first matching entry is then unset to be invalid, and the TCAM will subsequently ignore that entry. The TCAM then performs another lookup to report the second match. This process continues until there are no more matching results. Finally, all the valid bits are set to be valid for the next packet.

As analyzed in [55], identifying k matching results requires $7k$ cycles, as it takes a total of 6 cycles to invalidate and later revalidate an entry. Furthermore, all entries are searched k times. Hence, the energy consumption level is high when packets match many entries. In addition, the match latency is high because performing a serial match loses the benefits of parallelism.

4.2.3 MUD Scheme

Lakshminaryanan, et al., proposed a novel algorithm to support both range matching and multiple matching in TCAMs [55]. Their approach is based on the observation that some commercially available TCAMs have 144 bits per entry, while the 5-tuple typically used for packet classification has only 104 bits. They proposed a scheme called Multi-match Using Discriminators (MUD) as illustrated in Figure 4-3. The basic idea is to encode the index of the entry and include the encoded value in each TCAM entry. For example, for the first entry in a TCAM, attach 0001 (the value one in binary) after the filter in that entry, and for the

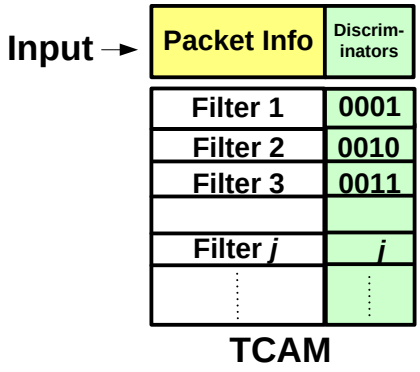
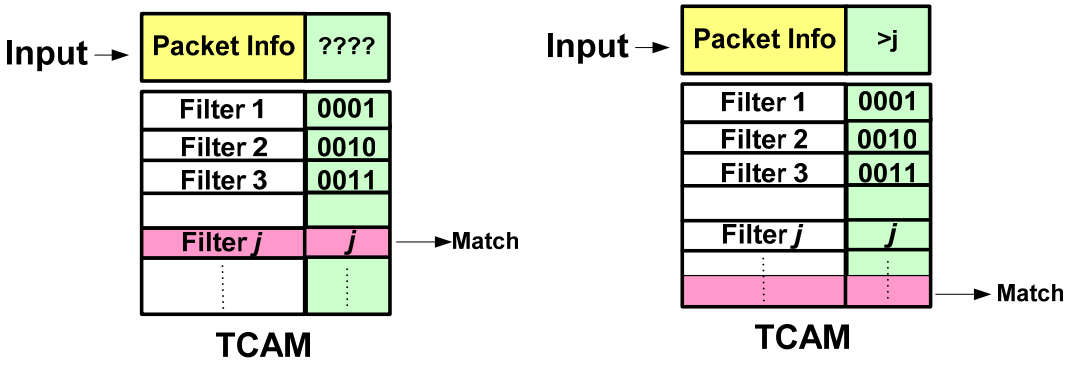


Figure 4-3. An illustration of the MUD scheme.



Initially the discriminators are set to don't cares — Set the discriminators as ">j"
First TCAM lookup matches the *j*th entry — Second TCAM lookup to get another match

Figure 4-4. An example of matching process with MUD scheme.

second entry, attach 0010 (the value two in binary). When searching for a match, MUD appends the input packet with a set of discriminators. The packet information is compared with the filters in parallel, while the discriminators are compared with the encoding of the indexes in parallel. Figure 4-4 shows an example of the MUD scheme. Initially, discriminators are all set to "don't care" states, meaning the input packet can match results at any index. After finding a matching result at index *j,* the TCAM is searched again with a discriminator field value that is set 'greater than *j'* to get the second matching result. 'Greater than *j*' is a range, it cannot be directly supported by TCAMs because there are only three states (0, 1, or don't care) in the TCAM. Hence, the scheme needs to expand this range to

prefixes. Consequently, MUD may need multiple TCAM lookups to obtain the second matching result. The authors showed that MUD needs *1+d+(k-2)*(d-1)* TCAM lookups to get *k* matching results, where *d* is the logarithm of the number of entries in the TCAM. The worst case lookups can be decreased to *1+d*(k-1)*/r with Database-Independent Range Pre-Encoding DIRPE, where *r* (smaller than *d*) is a parameter controlled by the number of available bits in each TCAM entry [55].

MUD does not require any modification to existing TCAMs. Compared to commercial approaches, MUD does not need to store the per-search state in the TCAM (i.e., invalidating the previously matched TCAM entries) to get the multi-match results. Therefore, it performs well in multi-threaded environments. However, it shares the same problem with commercial approaches: the number of TCAM accesses needed per packet is linear in the number of matching results. We will show later in Section 4.4 that a packet can match up to 12 unique filters for the SNORT filter sets and thus requires a maximum of 20 TCAM lookups. This is the worst case performance. However, even in the common case a packet will frequently trigger many TCAM lookups. For example, a regular HTTP packet matches at least 4 unique filters. A Napster file-sharing packet can match 8 unique filters and thus requires a maximum of 15 TCAM lookups. In addition, all TCAM entries are accessed during each TCAM lookup, so the power consumption of MUD can be high.

4.2.4 Geometric Intersection Methods

The Geometric Intersection Method proposed in the previous chapter inserts filter intersections in the TCAM to handle a packet that matches multiple filters. These intersection filters and the original filters are inserted into the TCAM and compared in parallel with the input packet. Afterwards, the TCAM matching result can be used as an index into SRAM to get all the matching results as shown in Figure 3-2. Theoretically, the number of intersection

filters can grow exponentially in the number of fields in the filter. Real-world filter sets are typically simpler than the theoretical worst case, but we still observe that the SNORT rule header set creates intersections that are ten times the original filter set size [67 2004]. This approach is expensive both in memory and power consumption when the filter set has many intersections, as is the case for SNORT.

As we have shown in the above analysis, none of the previously proposed approaches is both memory and power efficient. Next in Section 4.3, we present a scheme that uses off-of-shelf TCAMs and meets both requirements with a Set Splitting Algorithm (SSA), which greatly reduces the need of storing intersection filters in the TCAMs and accessing each filter only once.

4.3 A Memory and Power Efficient Approach

In this section, we present our TCAM-based algorithm called Set Splitting Algorithm (SSA), which uses a small memory size and accesses each filter in the TCAM only once per packet. We start this section with examples to illustrate the intuition of SSA in Section 4.3.1 and we will present the detail of the algorithm in the subsequent sections.

4.3.1 Illustration of Our Approach with Examples

Our SSA algorithm is based on the Geometric Intersection Method, which requires only one TCAM lookup per packet. As an example, consider an input of two simple filters with only two fields (source and destination ports). The geometric presentation of the filters in a two dimensional space is shown in Figure 4-5 with the x axis as the source port and the y axis as the destination port. Note that, if a filter contains *F* fields, its geometric presentation will be in an *F* dimensional space. In this example, filter one (source port *any* and destination port *8080)* and filter two (source port *80* and destination port *any*) generates an intersection filter

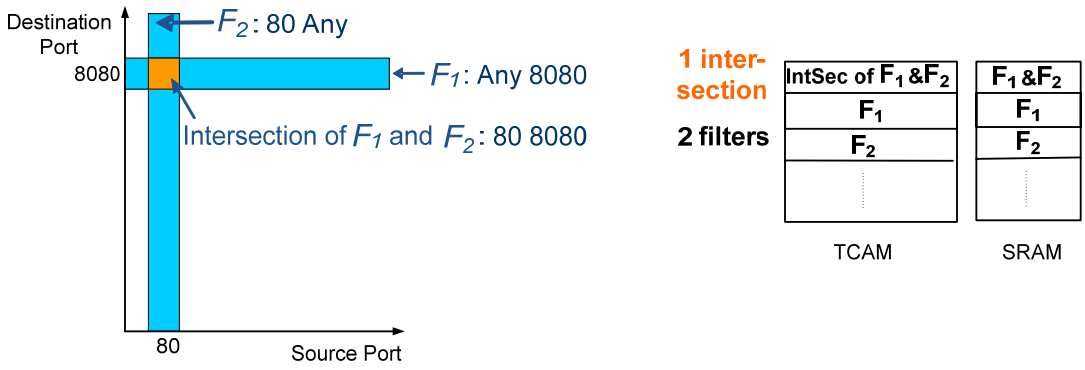
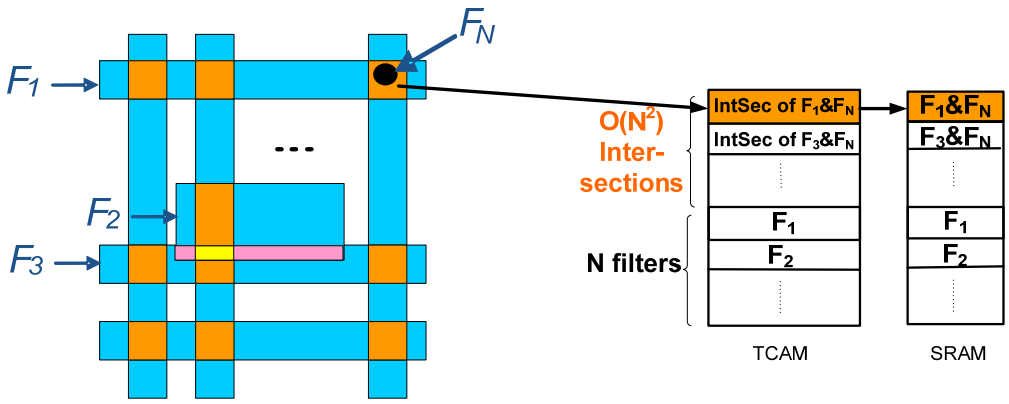


Figure 4-5. An example of two intersected filters.

(source port *80* and destination port *8080*). The Geometric Intersection approach, shown in the right side of Figure 4-5, needs to include this intersection filter at the top of the TCAM to capture packets matching both filters. With this intersection filter, the TCAM can report the multi-match classification result with just one TCAM lookup. This is a very simple example as there are two filters and they generate only one intersection. For some filter sets, this approach can create large numbers of intersections. A more complicated example is shown in Figure 4-6, where all the filters intersect with each other in a two dimensional space. For this case, we need to insert $O(N^2)$ intersection filters at the top of the TCAM for reporting the multiple matching results with just one lookup. For example, an input packet matching both F_1 and F_N (shown using a dot in Figure 4-6) will match the first intersection filter in the TCAM and hence be indexed to SRAM to get the matching result of F_1 and F_N.

Too many intersection filters result in high storage requirements and power consumption. This can be solved if we are willing to sacrifice time to reduce space and power. Take the previous example in Figure 4-6. If we split the filters into two groups, we can check those two groups separately and report the matching results from both groups respectively (Figure 4-7). Splitting filters into multiple sets yields a very nice property: intersections generated by

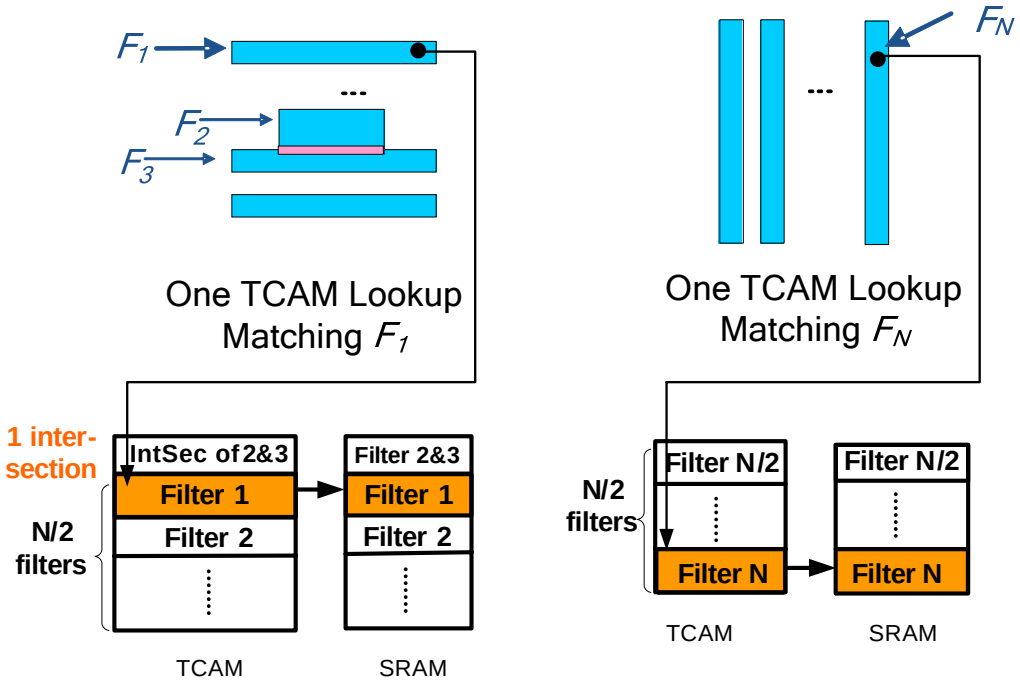
filters of *different* groups are no longer needed to be stored in TCAM. This is because the intersection filters are used to report multiple matching results with one TCAM lookup, e.g.,



Using Geometric Intersection Method, include all intersections in the TCAM
Storage cost: *N* filters +$O(N^2)$ intersection
Classification speed: 1 TCAM lookup time

Figure 4-6. An example of *N* filters.

F_1 and F_N. When filters are spread over different sets and we conduct separate TCAM lookups into these sets, the matching of these filters would be reported separately as shown Figure 4-7. As a result, we don't need to keep such intersection filters. However, we still need to keep intersections that are generated by filters in the *same* set because we only conduct one TCAM lookup into each set. Hence, intersection filters are still necessary to report matching of multiple filters case in the same set. For instance, in Figure 4-7, the intersection of filter F_2 and F_3 still needs to be included in the TCAM to report the matching of both F_2 and F_3 with one TCAM lookup. As shown in Figure 4-7, by dividing the filter set into two groups, most intersections are caused by filters from different groups, so they can be removed. We can reduce the number of TCAM entries from $O(N^2)$ to $N+1$, but it costs two TCAM lookups.



Divide filters into two sets, perform two TCAM lookups
Storage cost: *N* filters +1 intersection *(F_2* and *F_3)*
Classification speed: 2 TCAM lookups time

Figure 4-7. Separate filters into two sets.

The main idea behind splitting filters into multiple sets is to reduce the number of intersections filters needed in the TCAM. As we have shown above, intersections from different sets are no longer needed in TCAM. Hence, if filter sets are split in a way so that most intersections occur between filters in different sets, we can save TCAM space and consequently also reduce power consumption. In addition, every set is accessed only once per packet and therefore the algorithm is power efficient. Note that although filters are separated into two sets, we do not necessarily require two TCAMs. TCAM vendors now provide a blocking feature (introduced in Section 2.3.2) that divides a TCAM into several blocks and allows users to selectively search one or several blocks in parallel. With this feature, different sets of filters can be put into different blocks of the same TCAM and be accessed separately. Since the sets generated by SSA are logically independent, these lookups can even be pipelined to achieve a higher rate.

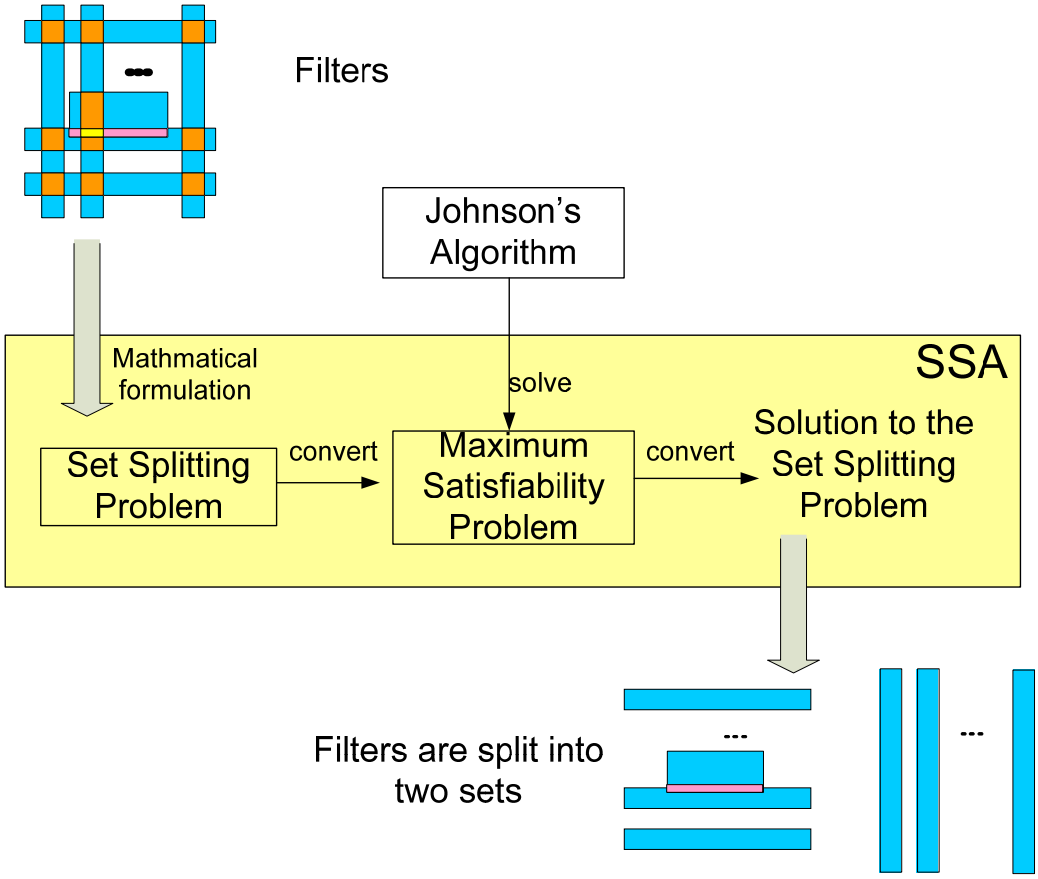


Figure 4-8. An overview of SSA.

Next, we present our Set Splitting Algorithm (SSA) that can automatically separate filters into multiple sets. Figure 4-8 illustrates the flow chart of SSA. Given a set of filters, it first mathematically formulates the problem into a set splitting problem. Since the set splitting problem is NP hard, SSA converts it into a maximum satisfiablity problem which has an efficient approximation algorithm – Johnson's algorithm [71]. SSA uses Johnson's algorithm as a subroutine to solve the maximum satisfiablity problem and then converts the solution back to original problem to get the final splitting results. SSA guarantees the removal of at least half of the intersections each time the filter set is split in two. In the remainder of this section, we will follow this flow chart and first present the mathematical formulation of the problem in Section 4.3.2. Then we quickly review Johnson's algorithm in Section 4.3.3 and finally present our SSA approach in detail in Section 4.3.4.

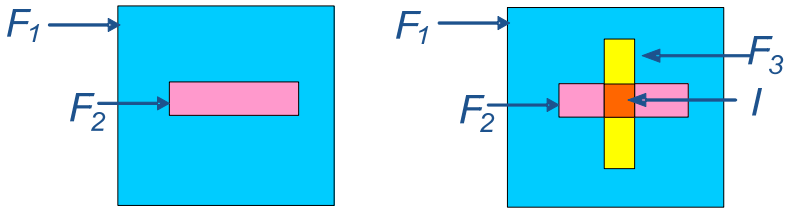


Figure 4-9. Example of filter intersections.

4.3.2 Mathematical Formulation

The previous section proposed an approach to split filters into multiple groups to decrease the number of intersection filters. In this section, we formulate the filter splitting problem as follows.

Suppose there are N filters $F_1, F_2, ..., F_N$, which create M intersections $I_1, I_2, ..., I_M$. For example, I_1 = *intersection of* (F_1, F_5, F_6). We only consider the *overlapping intersections* that are different from the original filters because these intersections consume extra TCAM entries. For the intersections that are the same as one of the original filters, we call them *contained intersections*. For example, in Figure 4-9, although F_1 and F_2 overlap, the intersection I is the same as F_2 ($I = F_1 \cap F_2 = F_2$). This intersection is a contained intersection and doesn't need to be present in the TCAM because when a packet matches F_2, we can simply report a matching of both F_2 and F_1. When filters create overlapping intersections, e.g., the F_2 and F_3 in the right hand size of Figure 4-9 intersect to create an overlapping intersection I ($I = F_2 \cap F_3$), we do need to add I into the TCAM because the intersection I is different from F_2 and F_3 ($I \neq$ either F_2 or F_3), thus requiring an extra TCAM entry. F_1 doesn't contribute to the generation of this intersection, so I is the intersection of F_2 and F_3 only. In other words, I can be eliminated only when F_2 and F_3 are in different sets. Throughout the reminder of this chapter, we use the term *intersection* to denote overlapping intersections.

We want to separate the filters into several sets so that filters generating intersections are in different sets. We define the residual intersection set I' as the overlapping intersections

generated by filters in the same set. Our objective is to separate filters into a minimum number of sets that satisfy $N + |I'| <$ TCAM size, which is an NP hard problem.

Suppose we restrict the problem by dividing the filters into two rather than multiple sets. Our new goal then would be to find a way to separate the filters into two sets so that number of residual intersections is minimal. Unfortunately, this problem is still NP hard and is known as the *maximum set splitting* or *maximum hypergraph cut* problem [72].

The best known approximation algorithm for the maximum set splitting problem yields a performance ratio of at least 0.72 to the optimal solution [73]. Here the performance ratio is defined as the number of clauses satisfied by the approximation algorithm to the optimal solution. This approximation algorithm requires quadratic programming and scales poorly. For example, [74] reports that it takes 27.35 seconds to compute the results for a set containing 50 instances (in our problem, a filter is an instance). When the set size increases to 100, it takes 355 seconds to compute the results. When the set reaches 200, it requires 10709 seconds using a Sun SPARCstation1 machine [75]. Although the latest Version of Pentium 4 processors [75] are 190 times faster, this scheme is still too slow for multi-match classification where there are hundreds to thousands of filters in the filter sets [10, 55]. In this chapter, we develop an efficient algorithm SSA to quickly divide filters into multiple sets to reduce the need of including intersection filters. SSA guarantees the removal of at least half of the intersections each time the filter set is split into two. In addition, it has a low complexity of $O(NM)$, where N is the total number of filters, and M is the total number of intersections. In reality, SSA removes almost all intersections, yielding a solution close to the optimal solution (removing all intersections) as we will demonstrate later in Section 4.4.2.

As previously explained in Figure 4-8, after mathematically formulating the problem, our SSA algorithm converts the set splitting problem into a maximum satisfiablity problem because it has a fast approximate algorithm -- Johnson's algorithm [71]. SSA uses Johnson's

algorithm to get an approximate solution to the converted maximum satisfiability problem, and finally maps the approximate solution back to the set splitting problem. In the next section, we briefly review Johnson's algorithm.

4.3.3 Johnson's Algorithm for the Maximum Satisfiability Problem

Our SSA algorithm converts the set splitting problem into a maximum satisfiablity problem. A maximum satisfiablity problem is defined as follows. Let L be N literal pairs $L = \{\{F_1, \overline{F_1}\}, \{F_2, \overline{F_2}\},.., \{F_N, \overline{F_N}\}\}$. Each literal can take either a true value or a false value. There are M clauses, with each clause consisting of a subset of literals either in a positive or negative form, e.g., $C_1 = \{F_1 \vee \overline{F_2} \vee F_6\}$. The goal is to find an assignment of L that satisfies the maximum number of clauses. Define K as the minimum number of literals in each clause. For $K \geq 2$, this problem is known to be NP complete [72].

Johnson's algorithm is an approximation algorithm for the maximum satisfiability problem. As presented in the pseudo code below in Figure 4-10, it works as follows: Initially, it assigns each clause C a weight = $2^{-|C|}$, where $|C|$ denotes the number of literals in C. For example, the weight of $C_1 = \{F_1 \vee \overline{F_2} \vee F_6\}$ is 2^{-3}. Next, the algorithm examines the literals one by one. For any literal F_i that has not been assigned a value, if the weight of all clauses containing F_i is higher than the clauses containing $\overline{F_i}$, assign F_i a true value. Now all the clauses containing F_i are all satisfied and hence can be removed. Clauses containing $\overline{F_i}$ are not satisfied yet, so we want the other literals within the clauses to have a higher probability of being selected as true later. Hence, the algorithm multiplies the weight of all the clauses containing $\overline{F_i}$ by 2. For the case where the weight of all clauses containing $\overline{F_i}$ is higher than those containing F_i, the algorithm performs the opposite actions.

Johnson's algorithm is simple, with *O(NM)* complexity. [71] proves that Johnson's algorithm can satisfy at least the fraction $(2^K-1)/2^K$ of the total clauses. For instance, when $K = 2$, it can satisfies at least ¾ of the clauses. Johnson's algorithm achieves the best approximate bound for $K>2$ [71].

Assign each clause with $|C|$ literals a weight = $2^{-|C|}$;
While (not all literals assigned weight yet){
Pick any remaining literal F_i;
    If the total weight of clauses containing F_i > those containing $\overline{F_i}$ {
        Assign F_i a true value;
        Remove all clauses containing F_i ;
        Double the weight of clauses containing $\overline{F_i}$;
    }**else**{
        Assign F_i a false value;
        Remove all clauses containing $\overline{F_i}$;
        Double the weight of clauses containing F_i
    }
}

Figure 4-10. Johnson's algorithm.

4.3.4 Set Splitting Algorithm (SSA)

Having now explained Johnson's algorithm, in this section, we present SSA, a memory and power efficient algorithm for multi-match classification. As illustrated in Figure 4-8, SSA first formulates the problem into a set splitting problem using the techniques explained in Section 4.3.2. Then SSA converts the problem into a maximum satisfiability problem using the following approach.

Every filter corresponds to a literal in the maximum satisfiablity problem, where literals can either take a true or false value. A true value denotes this filter is put into *Set* S_t, while a

false value denotes *Set* S_f. For every intersection, two clauses are added into the clause set, one with all positive literals, and one with all negative literals. For example, if I_1 represents the intersection of F_1, F_2 and F_6, add two clauses:

$C = \{F_1 \vee F_2 \vee F_6\}$ and $C' = \{\overline{F_1} \vee \overline{F_2} \vee \overline{F_6}\}$.

Clause C with all positive values denotes that at least one of the filters is in Set S_t. Similarly, Clause C' denotes that at least one of them is in Set S_f. If both clauses are satisfied, then at least one of the filters is in Set S_t and another one is in Set S_f. We will prove later, through a lemma, that if both clauses are satisfied then this intersection no longer need be present in TCAM. The total number of clauses is $2M$, where M is the number of intersections. After converting the set splitting problem into this satisfiability problem, SSA runs Johnson's algorithm to solve it and assign each filter F_i either a true or a false value. Below is the pseudo code of SSA.

Reduce the filter set splitting problem into a max satisfiablity problem:

Each filter F_i corresponds to a literal

For each intersection I_i generated by j filters: F_{x1}, F_{x2}, …, F_{xj}, add two clauses:

C = { $F_{x1} \vee F_{x2} \vee, \ldots, \vee F_{xj}$)

C' = ($\overline{F_{x1}} \vee \overline{F_{x2}} \vee, \ldots, \vee \overline{F_{xj}}$)

Run Johnson's algorithm to assign each filter F_i a true or false value

Put F_i in Set S_t if it is true.

Put F_i in Set S_f if it is negative.

Figure 4-11. Set Splitting Algorithm (SSA).

Next we prove that SSA has a nice property: it guarantees the removal of half of the intersections when a filter is split into two sets. We begin with a lemma and then prove this property through a theorem.

Lemma: If both clauses of an intersection are satisfied, this intersection is no longer needed in the TCAM.

Proof: Suppose I_1, the intersection of F_1, F_2, and F_6, has both clauses $C = \{F_1 \vee F_2 \vee F_6\}$ and $C' = \{\overline{F_1} \vee \overline{F_2} \vee \overline{F_6}\}$ satisfied. This means that at least one of (F_1, F_2, F_6) is true and one of them is false. According to the algorithm, these filters are split into different sets. Thus this intersection does not need to be represented in the TCAM.□

Theorem: SSA can remove at least 50% of the intersections each time the filter set is split into two sets.

Proof: Each clause is generated by an intersection; hence, it has at least two literals. In other words, $K >= 2$. From Johnson's results, at least $(2^K-1)/2^K = ¾$ of the clauses are satisfied. There are a total of $2M$ clauses, which means $2M*3/4 = 1.5M$ of the clauses are satisfied. Since there are M intersections, each corresponding to two clauses, at least $0.5M$ of the intersections have both clauses satisfied and hence can be removed according to the lemma. □

If we want to split the filter set further into more subsets (e.g., from 2 sets to 4 sets), this result still holds. Whenever the filter set is split, we can decrease at least 50% of the intersections. For example, if we split the filters into two sets, and then split both sets again, we can decrease the number of intersections to less than ¼ of the original amount.

The complexity of SSA is the same as Johnson's algorithm: $O(NM)$. Note the analysis above does not impose any restriction on the order for those literals examined while running Johnson's algorithm. In our simulation, we select literals based on the ratio of positive weight (total weight of clauses containing positive literals) to negative weight because we want to select literal that can satisfy the most clauses first. This results in a complexity of $O(NM+N^2)$.

4.4 Simulation Results

To demonstrate the effectiveness of SSA, we compare it with two well-known TCAM-based approaches: MUD and our Geometric Intersection Method presented in the previous chapter.

Table 4-1. Total number of unique SORT rule header size.

SNORT Version	Rules
2.0.0	240
2.0.1	255
2.1.0	257
2.1.1	263

We begin this section with evaluation methodologies we used in Section 4.4.1. We use the SNORT [76] rule headers to benchmark the three schemes and present the simulation results in Section 4.4.2. Since the SNORT rule header sets are fairly small, we also test our algorithms on synthesized larger filter sets in Section 4.4.3. We will show that SSA uses approximately the same amount of TCAM memory as MUD, but yields a 75% to 95% reduction in power consumption. Compared with the Geometric Intersection Method, SSA uses 90% less TCAM memory and power at the cost of one additional TCAM lookup per packet.

4.4.1 Evaluation Methodologies

We use the SNORT rule header set to evaluate performance. The unique rule headers in each version vary from 240 to 257 as we showed previously in Table 4-1. Since the SNORT rule header sets are fairly small, to test the scalability of our algorithm, we generate large synthesized sets as follows. We take a real-world single-match classification set used in a core router (3060 filters) [44] and insert new filters that intersect with the existing filters to create multi-match filter sets. As in the SNORT rule header set, filters in this single match classification also have five fields (protocol, source IP, source port, destination IP, and destination port). The newly inserted synthesized filters also have five fields. Each field in the synthesized filter is randomly selected from corresponding fields in old filter sets according to their appearance frequencies. Since the five fields in a new filter are randomly selected and then combined together to create a new filter, this filter is likely to intersect with

the old filters to create new intersections. The performance of different algorithms is largely dependent on the number of intersections. Therefore, we test our algorithms using different intersection rates, defined as the percentage of newly inserted filters to the size of the original filter set. We start at 5%, which constitutes around 150 new filters, and increase to 100%, meaning that the number of newly inserted filters has the same size as the original filter set (3060 filters).

We study the tradeoffs between different approaches on different data sets in terms of both memory and power consumption. We also test classification speed and update costs because multi-match classification solution must be fast enough to keep up with the increasing line rates. Next we explain these four metrics in detail.

Memory consumption. We use the total number of TCAM entries to reflect memory consumption because all entries have the same width (e.g., 144 bits). The total TCAM memory consumption is the number of TCAM entries used times the TCAM width.

Power consumption. Figure 2-6 of Section 2.3.2 shows that the energy used by a TCAM grows linearly with the number of entries searched in parallel and also with the number of TCAM accesses. Hence, we use the total TCAM entries accessed per packet as a metric for power consumption. It is defined as the product of the number of TCAM entries accessed per lookup and the number of lookups per packet.

Classification Speed. Memory lookup is usually the bottleneck of a packet classification system. For the three TCAM-based approaches (SSA, MUD and Geometric Intersection Method), the number of SRAM accesses per packet is equal to the number of TCAM accesses. This is because the TCAM output can be used as an index to fetch the results stored in SRAM. Hence, we report the maximum number of TCAM lookups per packet to reflect the worst case classification rate.

Table 4-2. Number of extra intersections filters in TCAMs.

Version	MUD	Geometric Intersection	SSA-2		SSA-4	
			Extra Inter-sections	Saving	Extra Inter-sections	Saving
2.0.0	0	3453	46	98.67%	1	99.97%
2.0.1	0	3754	47	98.75%	1	99.97%
2.1.0	0	3758	47	98.75%	0	100%
2.1.1	0	4067	55	98.65%	0	100%

Update costs. We randomly select 90% of the filter set as the base filter set and use the remaining 10% as the update filters. We test the insertion cost in terms of the number of newly inserted filters because the deletion of old filters is easier in TCAM-based approaches (simply mask those entries out). Inserting a filter into a first match TCAM may require moving the existing filters. There are existing approaches that prepare empty entries in TCAMs, or associate priority [76] or extra circuits with each entry [77]. Here, we only concentrate on the number of newly inserted filters as the metric of update cost.

4.4.2 Results on the SNORT Rule Header Set

We conduct tests on the SNORT rule header set and present the experimental results in this section. We will show the results on the synthesized filter sets later in Section 4.4.3. For the SNORT rule set, we test all the Versions of SNORT after 2.0. Table 4-1 shows that the unique rule headers (classification filters) in each Version vary from 240 to 257. We compare SSA against our previous Geometric Intersection Method and MUD in the next four subsections using the metrics of memory consumption, power consumption, classification speed, and update cost respectively.

Table 4-3. Total number of TCAM entries used.

Version	MUD	Geometric Intersection	SSA-2	SSA-4
2.0.0	240	3693	286	241
2.0.1	255	4009	302	256
2.1.0	257	4015	304	257
2.1.1	263	4330	318	263

4.4.2.1 TCAM Memory Consumption

The Geometric Intersection Method inserts all the intersections into the TCAM. The second column in Table 4-2 records the number of extra intersections (above and beyond the original filters) that must be included. Although it is well below the theoretical upper bound $O(N^F)$, it is still roughly 10 times the size of the original filter set. After applying SSA to divide the filters into two sets (SSA-2), the number of extra filters that need to be included in the TCAM falls below 55, *removing more than 98% of the intersections*. When splitting the filters into four sets (SSA-4), SSA almost removes the need to include any extra filters due to intersections. Note that the SNORT rule set contains range and negation (fields, such as "not port 80"). These filters may need to be mapped into multiple TCAM entries. There are many existing approaches for dealing with range and negation. For example, we can use extra encodings [55, 61] or hardware circuits [66] to solve the range problem. Furthermore, the negation removing scheme proposed in the previous chapter can be used to efficiently map negations into TCAMs. We assume these known techniques are used to handle range and negation filters. This assumption won't affect our comparisons because they all face the same percentage of filter size increase.

Table 4-3 shows the total number of TCAM entries required by the three approaches. MUD uses the smallest number of TCAM entries: just the number of filters. The Geometric Intersection Method requires a very large number of intersection filters. Hence, it consumes

the largest number of TCAM entries. SSA dramatically removes the extra intersection filters that need to be inserted into the TCAM. The number of TCAM entries needed is extremely close to the MUD scheme. Although MUD is optimal in the memory cost, it needs to access TCAM multiple times to get all matching results. Hence, the classification speed of MUD is not as high as SSA, as explained in the next subsection.

4.4.2.2 Classification Speed

For a given packet, only one TCAM lookup is required by the Geometric Intersection Method. For SSA, the number of lookups may be larger than one, yet remains deterministic. If the filter sets are split into two sets, two separate TCAM lookups are needed. These two TCAM lookups access different sets of filters. Because no logical relationship exists between them, these two lookup processes can be fully parallelized. If filter sets are split four ways, four TCAM lookups are needed. In the current Internet, packet sizes vary from 40 bytes (TCP packets with no payloads) to 1500 bytes (Ethernet packets) [78]. [45] reports that the average packet size is 402.7 bytes. If one packet requires four TCAM lookups of 4 ns per lookup, SSA can achieve a 201.35 Gbps classification rate. In the worst case where packets are 40 bytes, SSA can still achieve a 20 Gbps rate, which is well above the line rate of the current Internet backbone.

For the MUD approach, the number of TCAM lookups is related to the number of filters a packet matches. Packets that don't match any filters need only one TCAM lookup, while a packet that matches k filters requires $O(k)$ lookups because each lookup can report at most one matching result. In addition, multiple TCAM lookups may be required even for reporting one matching result, because it needs to represent range using the three states of the TCAM. [55] shows that MUD takes $1+d*k/r$, for reporting k matching using MUD, where d is the logarithm of the total number of filters and r is a parameter for encoding, which is decided by

Table 4-4. Update cost in terms of newly inserted filters.

Version	MUD	Geometric Intersection		SSA-2		SSA-4	
		Avg	Max	Avg	Max	Avg	Max
2.0.0	1	31.73	157	1.33	17	1.002	2
2.0.1	1	35.24	135	1.34	19	1	1
2.1.0	1	34.71	135	1.36	20	1.002	2
2.1.1	1	36.00	172	1.41	26	1.006	2

the number of available bits in each TCAM entry. For the SNORT filter sets, d is 8 or 9 and r can be 5 or less. One packet can match a maximum of $k = 12$ filters. For such packets, MUD takes *at least* 12 TCAM lookups to get all the matching results. The *worst case* number of TCAM lookups ($1+d*(12-1)/r$) can be as high as 20 using MUD.

The above analysis is the worst case performance of MUD. Since the number of TCAM lookups is workload dependent, we looked at some high frequency packets. A HTTP packet matches at least 4 unique filters and thus requires 5 to 9 TCAM lookups. A Napster file-sharing packet can match 8 unique filters and thus requires 9 to 15 TCAM lookups.

4.4.2.3 Update Cost

The update cost for the Geometric Intersection Method is high, as newly inserted filters may intersect with the existing filters and may result in many insertion operations. Table 4-4 shows the average update costs per newly inserted filter in terms of the number of newly generated filters (including intersection filers) in the TCAM. The third column shows the maximum number of insertions. One filter can generate up to 157 new TCAM entry insertions. The Geometric Intersection Method is obviously slow during the update process.

The number of newly inserted filters is always 1 for MUD, as it does not need to consider any intersections. The average insertion cost decreases to almost 1 when using SSA-2. However, the max update cost is still high (around 20). If the filter set is split into four sets

Table 4-5. TCAM entries accessed per packet.

	MUD (Worst case)	MUD (HTTP Packets)	MUD (Napster Packets)	Geometric Intersection Method	SSA-2	SSA-4
2.0.0	4800	1200	2160	3693	279	233
2.0.1	5100	1275	2295	4009	292	245
2.1.0	5140	1285	2313	4015	295	247
2.1.1	5260	1315	2367	4330	309	255

(SSA-4), the number of newly inserted filters is reduced to approximately one and the worst case cost is two, which is similar to MUD.

4.4.2.4 Power Consumption

We mentioned in Section 4.4.1, the energy used by a TCAM grows linearly with the number of entries searched in parallel and also to the number of TCAM accesses. Hence, we use the product of the number of TCAM entries accessed per lookup and the number of lookups per packet as a metric for power consumption.

The power consumption of the MUD-based approach is related to the incoming packet rates. As we explained in Section 4.4.2.2, in the worst case, a packet may need up to 20 TCAM accesses to get all the matching results. For each access, all the entries in the TCAM are compared in parallel. Hence, the total number of TCAM entries accessed is 20 times the filter set size in the worst case, which is over 20*240 = 4800 entries as shown in the first column of Table 4-5. If all the packets are HTTP packets, then each packet needs at least 5 TCAM lookups, meaning that it accesses at least 5*240 = 1200 entries for one packet. A Napster packet needs at least 9 TCAM lookups and hence accesses at least 9*240 = 2160 entries.

The Geometric Intersection Method only performs one TCAM lookup. Therefore, the number of TCAM entries accessed per packet is the same as the number of TCAM entries used. This value is around 4000 due to large number of intersections included in the TCAM.

Table 4-6. Total number of extra intersections filters in TCAMs.

Insertion Factor	Geometric Intersection	SSA-2		SSA-4	
		Intersections	Saving	Intersections	Saving
0	359	22	93.87%	2	99.44%
0.05	418	20	95.22%	1	99.76%
0.1	488	45	90.78%	3	99.39%
0.2	733	52	92.91%	4	99.45%
0.3	1080	112	89.63%	1	99.91%
0.4	1312	78	94.05%	9	99.31%
0.6	2086	171	91.80%	9	99.57%
0.8	2488	208	91.64%	4	99.84%
1	2883	229	92.06%	7	99.76%

For SSA, although we perform several TCAM lookups, they look into different groups of filters (stored in different TCAM blocks or different TCAMs). Each TCAM entry is accessed only once per packet. Hence, the total number of TCAM entries accessed is just the number of entries in the TCAMs (less than 300). The energy used by SSA-4 and SSA-2 is similar as shown in the last two columns in Table 4-5. SSA-2 saves at least 90% of the energy consumed by the Geometric Intersection Method. Compared to MUD, it saves over 95% when compared to the worst case performance, and saves 76% for HTTP packets and 87% for Napster file-sharing packets.

4.4.3 Results on a Synthesized Multi-match Filter Set

The SNORT rule header set is relatively small. To test the algorithms on larger filter sets, we generate large synthetic multi-match classification test sets as described in Section 4.4.1. We take a real-world single-match classification set used in a core router with 3060 filters each with 5 fields, and insert new filters (also with 5 fields) that intersect with the existing filters into it. We test our algorithms using different intersection rates, defined as the percentage of newly inserted filters to the size of the original filter set. We start at 5%, which constitutes

around 150 new filters, and increase to 100%, meaning that the number of newly inserted filters is the same size as the original filter set size (3060 filters).

Although the synthesized filter sets are larger than the SNORT rule sets, they generate relatively fewer intersections, as shown in the second column of Table 4-6. SSA also works well for these large filter sets. Splitting the filter set into two sets removes over 90% of the extra intersections, while splitting the set into four sets almost eliminates the need to include extra intersections.

The results on synthesized filter sets on the classification speed and energy consumption are similar to those of the SNORT database. SSA-2 requires only two memory lookups per packet. However, MUD can match 12 filters in the worst case and thus requires up to 20 TCAM accesses per packet. Hence, SSA-2 is faster and more energy efficient than MUD.

4.5 Conclusions

Multi-match packet classification is a key packet-processing operation needed in important applications such as network intrusion detection and packet-level accounting. Previously proposed TCAM-based approaches suffer from high power consumption or high memory usage. In this chapter, we develop a new set splitting algorithm (SSA) that reduces the TCAM memory and power consumption by 90% when tested on the SNORT rule set. This is accomplished by using a novel scheme that splits filters into multiple TCAM blocks such that, for each split, the number of overlapping rules within each TCAM block is reduced by a factor of at least two. The benefits of SSA are summarized as follows:

Low Memory Usage. We showed that it is not necessary to include the intersections caused by filters of different sets in the TCAM. Each time a filter set is split into two sets, SSA guarantees the removal of at least 50% of the intersections.

Low Power Consumption. SSA uses a small amount of TCAM memory and accesses each TCAM entry once per packet. Hence, the power consumption is low.

Deterministic Lookup Rates. If SSA splits filters into k sets, then k TCAM lookups are needed. The number of TCAM lookups is independent of the input packet.

Supports Parallelism. The filter sets generated by SSA are uncorrelated. Thus, the lookups into these filter sets can be parallelized or pipelined.

Low Update Cost. Since filters are split into uncorrelated sets, the update cost is local to one set and all other sets remain the same.

This chapter and Chapter 3 presented packet inspection schemes suitable for handling the fixed fields of the packet header. Chapter 3 emphasizes developing a very high speed solution, while this chapter provides a more general and balanced solution taking into consideration three factors: memory size, classification speed and power consumption. We provided a solution for multi-match classification while doing it fast, with little power and few TCAM entries.

Besides packet header processing, as explained in Section 1.3, high speed deep packet inspection also involves a more difficult and less structured problem -- packet payload scanning. There are two types of payload matching requirements: fixed string matching and regular expression matching. We will present a solution for matching fixed strings using TCAMs in Chapter 5. In Chapter 6, we will study the typical regular expressions in payload scanning network applications and propose a fast regular expression matching scheme suitable for both single-core and multi-core processor based architectures.

5 Fixed String Pattern-Matching Using TCAMs

As explained in Chapter 1, high speed deep packet inspection involves two aspects: packet classification on the packet header, and pattern matching on the packet payload. For the first aspect, Chapters 3 and 4 proposed TCAM-based approaches that perform multi-match packet classification on the packet header. In this and the next chapter, we propose pattern matching algorithms for the packet payload.

Performing pattern-matching at high speeds is an extremely hard problem. In typical network applications like intrusion detection [10], there are thousands of patterns that must be matched against each packet's payload. Due to the wide variety of application formats, we often do not know apriori the *anchor location,* that is, the byte offset where the pattern starts in the packet. Hence, we need to check every byte of the packet payload, at the line speed of the network. In addition, these patterns are often highly complex. There are *fixed string patterns* with arbitrarily lengths; *correlated patterns* where it is necessary to check for certain pattern sequences; and *patterns with negation* where it is necessary to detect the absence of a pattern. Sometimes, it is infeasible to enumerate the pattern using a fixed list of strings, so *regular expressions* are used as a pattern language. We will explore the use of regular expression pattern matching in the next chapter. Although fixed strings patternscan be specified using regular expressions, we show later in Section 6.5.2 that correlated patterns are very hard to match using regular expression techniques. They make exponential interactions and generate large DFAs. Therefore, for these types of patterns, it is still better to identify them individually and then correlate the matching to find the composite patterns. TCAMs are good at finding these individual patterns. In this chapter, we propose a TCAM-based fixed strong

matching scheme. In particular, we propose algorithms for arbitrarily long patterns, correlated patterns, and patterns with negation.

The rest of the chapter is organized as follows. We motivate the chapter in Section 5.1 and review related work and show that no existing approaches can perform pattern-matching on thousands of complex patterns at high rates in Section 5.2. Then we study the typical patterns used in payload scanning applications and define generalized pattern formats based on the analysis of different signature sets in Section 5.3. These pattern formats are *short fixed strings* that are smaller than the TCAM width, *long fixed strings* that exceed the TCAM width, and c*omposite patterns* that are composed of a series of patterns. Sections 5.4 through 5.6 present our schemes to address these pattern formats respectively, with Section 5.4 addressing the short fixed strings, Section 5.5 presenting schemes for long patterns, and Section 5.6 providing algorithms for composite patterns. Section 5.7 analyzes our scheme and discusses strategies against malicious attacks. We present simulation studies on both the real world traces and synthesized worst case traffic in Section 5.8. For the ClamAV virus database [46] with 1768 patterns whose sizes vary from 6 bytes to 2189 bytes, the proposed scheme can operate at a 2 Gbps rate with only a 240 KB TCAM.

5.1 Introduction

Many applications such as HTTP load balancing and email SPAM filtering, require packet payload scanning. For example, network intrusion detection systems monitor packets in the network and scan packet payloads to detect malicious intrusions or Denial of Service (DoS) attacks. SNORT[10], a popular open source Network Intrusion Detection System (NIDS), has thousands of rules, each specifying a particular pattern representing a signature of an intrusion method. Besides a large number of patterns, another difficulty is that virus signature databases often have correlated patterns, which must match sequentially to indicate the

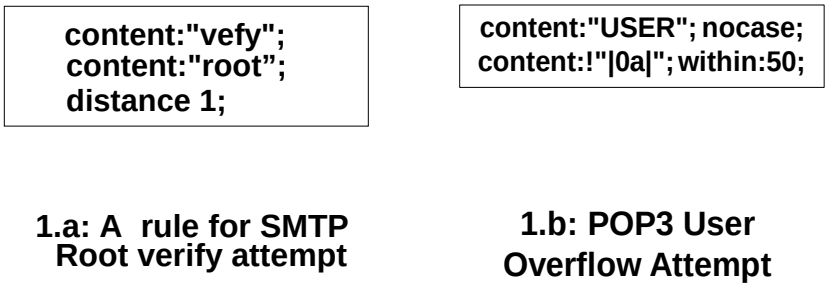


Figure 5-1. Two example patterns from SNORT rules.

presence of an attack. For example, Figure 5-1.a shows a rule for Simple Mail Transfer Protocol (SMTP) root verify attempt, which requires matching two patterns ("vefy" and "root") in sequence. The rule in Figure 5-1.b is another example where the system searches for the first pattern "USER". If it does not detect a return character (\n, |0a| in ASCII format) within the next 50 bytes, it will raise an intrusion alarm signaling an overflow attack attempt.

A large number of these complicated patterns make it hard for pure software-based pattern matching algorithms to keep up with network line rates. The SNORT system, for example, implements its pattern matching algorithms in software. It can handle link rates only up to 250Mbps [11] under normal traffic conditions even using SUN's special Security Defense Appliances [12] and has significantly poorer worst-case performance. These rates are not sufficient to meet the needs of even medium-speed access or edge networks. Since worms and viruses may possibly originate inside the network, NIDS are also required to scan packets inside the network, which is usually Gigabit rates or higher.

Building SNORT-like intrusion detection systems to run at Gigabit rates is a challenging task. Since patterns can appear anywhere in the packet payload, we need to examine every byte. The processing on every byte must complete within eight nanoseconds. Even with the latest Pentium processor (3.8 GHz) [79], eight nanoseconds of processing time yields just 30.4 CPU cycles. Clearly it is impossible to compare each incoming packet with thousands of patterns in a serial fashion. Moreover, since there are thousands of patterns, storing them all

in registers requires too much space. The majority will need to be kept in cache or the main memory. Eight nanoseconds is sufficient for only one or two memory accesses using fast Static Random Access Memory (SRAM) [80] for each byte in the packet payload. Multi-core processors can harness parallelism to speed up the pattern matching. However, the number of cores in today's state-of-the-art processors is limited. For example, the Intel network processor IXP2850 has 16 cores [52]. This approach cannot scale to the number of patterns (usually thousands) required by many networked applications. From the above analyses, we can see that purely software-based pattern matching schemes cannot process thousands of patterns at Gigabit and higher rates.

To perform high speed pattern matching, we must resort to hardware-based accelerators that have the ability to perform high speed parallel comparison. TCAMs meet this requirement and thus are a natural fit for the pattern matching problem. As introduced in Chapter 2.3.2, TCAMs are widely used for IP header based processing such as longest prefix match. Because of their intrinsic parallel search capability, they can also be used effectively for the pattern matching functions such as those for intrusion detection systems. However, TCAMs, because of their limited width, do impose limitations on the pattern length that can be directly matched. Also, there is no direct method to handle correlated patterns such as the example patterns shown in Figure 5-1. In this chapter, we develop algorithms that use TCAMs as a building block for both high-speed and fixed string pattern matching. Before we present the detail of our algorithms, we first review the related work on fast string matching in the next section.

5.2 Related Work

The research community has extensively studied the fixed string matching problems. In this section, we only discuss approaches that are relevant to the network payload scanning

problem. We first review representative software-based schemes and then discuss FPGA and Bloom filter based schemes that are amenable to hardware implementation. Finally we state the reasons for picking TCAMs as a candidate solution and related work that uses TCAMs for payload scanning. The main reason for picking TCAMs is that TCAMs have strong parallel comparison abilities.

5.2.1 Algorithms for Software-only Environments

The most influential software-only algorithms are: Knuth-Morris-Pratt(KMP) [81], Boyer-Moore [82], Aho-Corasick [83], and Commentz-Walter [84].

The KMP and Boyer-Moore algorithms work most efficiently for single pattern searching [23]. They build skip tables that record the characters that do not appear in the pattern. If we identify a character in the input that is in the skip table, we can skip the character and thus avoid backtracking. The search time for an m byte pattern in a n bytes packet is $O(n+m)$. If there are k patterns, the search time is $O(k(n+m))$, which grows linearly in k. Hence, this method is slow when there are thousands of patterns.

The Aho-Corasick and Commentz-Walter algorithms match multiple patterns simultaneously. They both pre-process the patterns and build a finite automaton that can process the input packet in $O(n)$ time. Although both algorithms are fast, they suffer from an exponential state requirement. One of the network intrusion detection systems, Bro [25], uses a similar deterministic finite automaton based approach. Bro generates so many states that only a part of the automaton is kept in memory. The system dynamically extends the automaton based on runtime information. This degrades the system performance.

Recently, new pattern matching algorithms specifically for content-based packet handling have been proposed. The Aho-Corasick-Boyer-Moore (AC_BM) algorithm proposed by Silicon Defense [25] combines the Boyer-Moore and Aho-Corasick algorithms. Another new

algorithm is the Setwise Boyer-Moore-Horspool algorithm by Fish, et al. [26], whose average case performance is better than Aho-Corasick and Boyer-Moore. These algorithms greatly improve SNORT's pattern matching speed (e.g., 250Mbps). However, their performance is still one order of magnitude less than the line rate needed for deployment in modern networks.

5.2.2 FPGA-based Approaches

FPGAs can be programmed for fast pattern matching due to their exploitation of reconfigurable hardware capability and their ability for parallelism. To search for a regular expression of length *n* on an FPGA, one approach is to build a Deterministic Finite Automaton (DFA) that requires $O(2^n)$ memory and takes O(1) time per text character. Sidhu, et al., proposed a Nondeterministic Finite Automaton (NFA) approach using FPGAs [49]. Their approach requires only $O(n^2)$ space and is still able to process each text character in *O(1)* time.

The above two approaches are optimized for single keyword searching and do not scale well for multiple patterns. The recent work by Baker, et al., uses a modified KMP algorithm [85]. Each pattern is still treated independently; however, multiple (100 reported in [85]) patterns can be pipelined at gigabit rates. The downside of this approach is that patterns are searched sequentially, so the overall latency increases proportionally with the number of patterns.

5.2.3 Bloom-filter-based Approaches

Dharmapurikar, et al., proposed a multiple-pattern matching approach using parallel Bloom filters [86]. Their approach can handle thousands of patterns. The proposed scheme builds a Bloom filter for each possible pattern length. This could impose parallelism limits in some

virus databases because pattern lengths vary from tens to thousands of bytes and there are hundreds of possible patterns lengths as we will show later in Section 5.8.2.

5.2.4 CAM-based Approaches

There are two types of CAMs, binary CAMs where each bit can only take one of the two states (0 or 1), and ternary CAMs (TCAM, introduced in Section 2.3.2) where each bit can take one of the three states (0, 1 and "don't care?"). In both Binary CAM and TCAMs, each entry is a bit vector of cells, where every cell can store one bit. Therefore, a CAM entry can be used to store a string. CAMs are built for parallel comparison of the input against a large number of entries. Hence, it is a good candidate for a solution to our multiple-pattern matching problem.

Gokhake, et al., proposed a pattern matching approach using binary CAM [87]. They restrict each pattern to be exactly w bytes and insert these patterns to the CAM, one pattern per entry. The space needed for k patterns each with w bytes is only kw bytes of CAM. Given an input, their approach first looks up the first w bytes of the payload in the CAM. Then it shifts one byte and does a CAM lookup with the second byte to get the next $w+1$ bytes of payload. This process is repeated until reaching the end of the payload. To search a packet of length n, the CAM-based approach can provide an answer in a deterministic time of $O(n+w)$ CAM lookups. One limitation of the proposed approach is that all patterns must have lengths equal to the CAM width. In this chapter, we develop algorithms for arbitrarily length patterns and other complex patterns, including correlated patterns and patterns with negations. In the next section, we present the definition of the complex patterns that we are going to address in this chapter.

Table 5-1. Patterns used in different systems.

	Total # of patterns	Simple fixed string patterns	Composite fixed string patterns	Regular expressions
SNORT (Version 2.1.2)	1693	1039	527	127
BRO (Version 0.8)	2870	0	0	2870
ClamAV (Version 0.15)	1768	1768	0	0
Linux layer 7	70	0	0	70

5.3 A study of Patterns and Problem Definition

We studied the typical patterns that appear in the network payload scanning applications including the SNORT intrusion detection system [10], Bro intrusion detection system [16], ClamAV anti-virus system [17], and the Linux Layer 7 filter [22]. The number of patterns in these systems varies from 70 to 400. We identify three types of patterns from these systems as shown in Table 5-1, namely *simple string patterns*, *composite string patterns*, and *regular expressions*. Next, we present the detailed definition of these pattern types. In particular, we study the simple fixed string patterns in Section 5.3.1, composite fixed string patterns in Section 5.3.2 and regular expression patterns in Section 5.3.3. In this chapter, we will present schemes for detecting the first two types of patterns. We will address the regular expression patterns in Chapter 6.

5.3.1 Simple Fixed String Patterns

Simple string patterns are quite common in network payload scanning applications. A simple fixed string pattern P of m bytes can be written as $P = b^1b^2 \ldots b^m$, where each b^i represents a byte. Sometimes, a fixed string pattern is associated with case sensitive or case insensitive options, where case insensitive distinguishes whether an upper case character or a lower case character, e.g., *a* or *A*, matches the pattern.

Fixed string patterns are very common in network payload scanning applications. For example, the ClamAV anti-virus system (Version 0.15) has 1768 simple patterns. The SNORT intrusion detection system (Version 2.1.2) contains 1693 patterns, of which 1039 are fixed string patterns. The fixed string patterns in these systems vary widely in length. In ClamAV, patterns range from 6 bytes to 2189 bytes, with an average of 55 bytes. In SNORT, patterns vary from one byte to 100 bytes, with many short patterns of one or four bytes. This large length variation breaks the assumption of some previously proposed approaches, such as the Bloom filters and the CAM-based methods mentioned in Section 5.2, that patterns have a limited number of lengths.

In addition to fixed string patterns, the SNORT filter rule set also has a large percentage of composite patterns. Next, we describe the structure of these composite patterns.

5.3.2 Composite Patterns

Simple patterns can be extended to form composite patterns. From SNORT [10], we identify the following two types of composite patterns:

1. Negation (!). *!P* denotes no appearance of pattern *P*.

2. Correlated patterns. If P_1 and P_2 are two patterns, $P_3 = P_1 .* P_2$ is a correlated pattern, where *"."* denotes arbitrary characters and *"*"* denotes any length. This pattern requires matching P_1 first, then some arbitrary content ".*", and finally matching pattern P_2. Note that ".*" can have infinite length, but in practice a length limitation is placed on it, e.g., equal to four bytes, or less than four bytes.

SNORT (Version 2.1.2) has 527 composite patterns, with 50 negation patterns and 477 correlated patterns. The former are usually for detecting buffer overflow attempts. For example, for a web login attempt, if we see the pattern "Authorization" with no return key "!\n" within the next 512 bytes, it is likely to be a buffer overflow attempt. On the other hand,

correlated patterns are often used to identify known application formats. Packets using a specific application protocol typically follow that application format and contain some user specific data. Therefore, correlated patterns may be expressed as a sequence of sub-patterns (application formats), while allowing distances between these sub-patterns to skip user specific data. For example, in Figure 5-1.a, four sub-patterns are used for detecting the MS SQL worm. In the SNORT rule set, the number of sub-patterns in one correlated pattern can go up to seven. None of the previously proposed schemes can handle these patterns at line rate. In Section 5.5, we will propose fast mechanisms to detect these patterns.

5.3.3 Regular Expressions

Besides fixed string patterns and composite patterns, the rest of the patterns in the four networking payload applications we studied are regular expressions. A regular expression describes a set of strings without enumerating them explicitly. We will present a more formal definition in the next chapter. Here we use an example to give a general sense of how these patterns are expressed. Consider a regular expression from the Linux L7-filter [1] for detecting Yahoo traffic: "^(*ymsg*|*ypns*|*yhoo*).?.?.?.?.?.?.?[*lwt*].**xc0**x80*". This pattern matches any packet payload that starts with *ymsg, ypns,* or *yhoo*, followed by seven or fewer arbitrary characters, and then a letter *l, w* or *t*, and some arbitrary characters, and finally the ASCII letters *c0* and *80* in the hexadecimal form.

Regular expressions are widely used for pattern matching due to their rich expressive power. In the Linux Application Protocol Classifier (L7-filter) [45], all protocol identifiers are expressed as regular expressions. Another intrusion detection system, Bro [46], also uses regular expressions as its pattern language. Recently, regular expressions are replacing explicit string patterns as the pattern matching language of choice in packet scanning applications. For example, SNORT [1] has evolved from a system with no regular

expressions in its ruleset in April 2003 (Version 2.0.0), to 127 in May 2004 (Version 2.1.2), and to 1131 regular expressions as of February 2006 (Version 2.4).

We will present algorithms for regular expression pattern matching in the next chapter. In this chapter, we propose fast pattern matching methods for the first two types of patterns, namely fixed string patterns and composite patterns. As discussed in Section 5.2.4, TCAMs are built for parallel comparison of thousands of entries and are an attractive candidate for fast pattern matching. Hence, next we propose TCAM-based methods for pattern matching. In particular, we propose algorithms for fixed strings that are shorter than the TCAM width in Section 5.4, for long patterns in Section 5.5, and for composite patterns in Section 5.6.

5.4 Short Fixed String Patterns

Let us first look at the simple case where all the patterns are simple deterministic patterns, with length shorter or equal to *w* bytes, where *w* is the width of the TCAM. Our approach is simply to place patterns into TCAM, with one pattern occupying one entry. If a pattern is shorter than *w* bytes, we pad it with "?" (don't care) bit as shown in Figure 5-2. For ease of explanation, in the rest of the chapter, we use alphabetic characters rather than binary forms as pattern examples. We assume that each character is one byte.

Patterns should be organized according to their lengths in descending order. This is because a TCAM only reports the first matching result and we want to identify all matching patterns. For example, if a pattern "ABC" is put in a lower index (top end in the example) in TCAM, matching of the "ABC" includes matching of a shorter prefix pattern "AB". If we place patterns in the other order, we cannot infer the matching of the longer pattern from matching the shorter pattern. Thus we may miss out some matching results.

The process of finding patterns in a packet is as follows: The first *w* bytes in the packet are mapped into TCAM (Figure 5-2.a). If there is a hit, we report the matched pattern. Next,

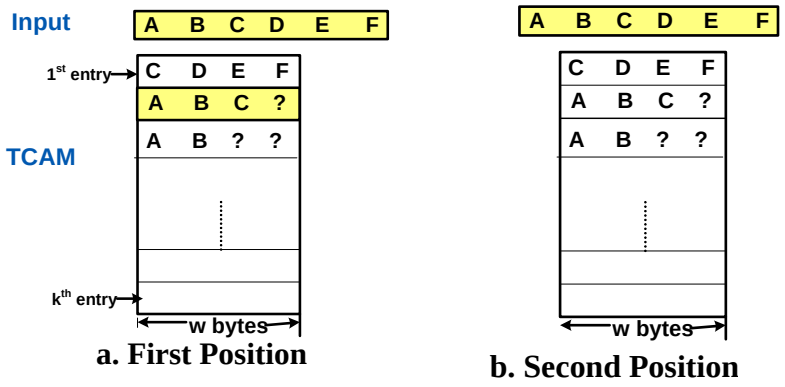


Figure 5-2. Scanning process.

shift one byte and check TCAM again as shown in Figure 5-2.b. This process is repeated until we have read the whole packet. Note that when we are at the end of the packet and the remaining packet size *t* is less than the TCAM width, we can pad it with all 0s and look it up in TCAM. However, only patterns less than *t* bytes should be reported as matches.

One TCAM lookup is needed for every byte position in the packet. Assuming the TCAM lookup time is 4 nanoseconds (ns), it can support a deterministic scan rate of 8 bits/4 ns = 2Gbps against thousands of patterns and is able to report all the matching patterns.

The scheme proposed in this section only applies to patterns that are shorter than the TCAM width. As we have shown in Section 5.3.1, patterns in network applications vary from several bytes to several thousands bytes and some of them may not be able to fit in one TCAM entry. In the next section, we present our methods for handling these long patterns.

5.5 Long Patterns

The previous section proposed a method for patterns that are shorter than the TCAM width. The TCAM width is configurable as we mentioned in Section 2.3.2. For example, a 1Mbyte TCAM can be programmed as 64K entries with 16 bytes per entry, or 1K entry with 1K bytes per entry etc. Given a pattern sets with variable lengths, one choice is to configure the TCAM width to be greater or equal to the longest pattern length and pad short patterns with

the ‘do not care’ states to reach the TCAM width. However this wastes precious TCAM resources when the pattern set has a large variation in pattern length, as in the ClamAV pattern set. The relatively small size and high cost of TCAM makes this a very inefficient approach.

Instead, our approach divides long patterns into shorter patterns (more patterns of shorter lengths) to save TCAM space, thereby allowing the TCAM width to be smaller. The overall process of matching long patterns is shown in Figure 5-3. Given a set of patterns, we first perform pre-processing. We divide long patterns into *sub-patterns* and store them in the TCAM. Then, we build a pattern table to record the actions upon matching of these sub-patterns. We also set up a matching table to store correlations between sub-patterns, so that we can re-link them together later to report matching of a long pattern. Given a packet, we take the first w bytes from it, where w is the TCAM width, and do a TCAM lookup. If there is no match, we shift one byte and perform another TCAM lookup. If we match a sub-pattern at the i^{th} position of the packet, we record it in a table in a memory data structure called the partial hit list. Later, if we match a sub-pattern at position $i+j$ $(0 < j <= w)$, we check the matching table to see whether the concatenation of this sub-pattern and a previously matched one forms a long pattern. If so, we report a matching of the long pattern.

The rest of the section is organized according to the flowchart in Figure 5-3. There are many ways to divide long patterns into smaller patterns. Next, in Section 5.5.1, we explain the tradeoffs of different pattern division methods and then present our division algorithm. After dividing a long pattern into sub-patterns and inserting them into the TCAM, we get the TCAM match results of these sub-patterns separately. Next, we need to link all the related match results together to report the matching of a long pattern. To achieve this, we keep a set of tables (pattern table, matching table, and partial hit list as shown in the grey boxes of Figure 5-3) in the memory to record the matching sub-patterns and relationship between

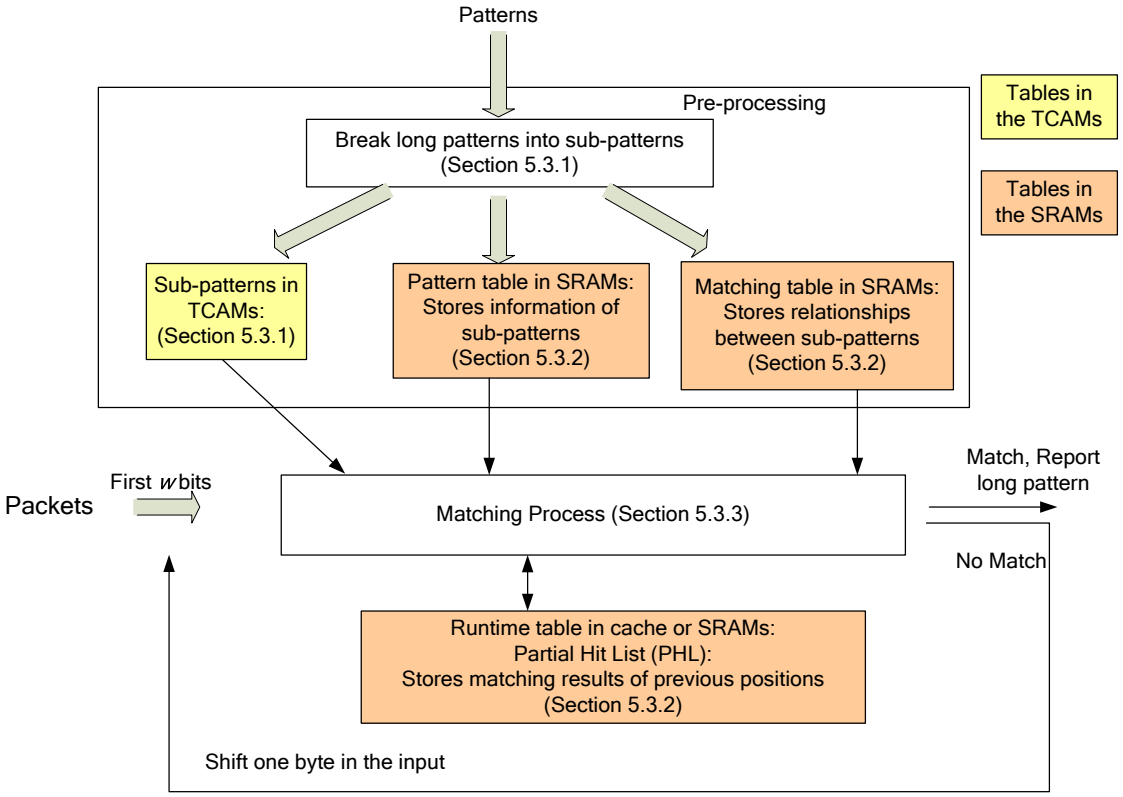


Figure 5-3. Flowchart of long pattern matching algorithm.

these them. These tables are explained in detail in Subsection 5.5.2 and we explain the algorithm for linking sub-pattern matching results to infer a matching of long patterns in Section 5.5.3.

5.5.1 Divide Long Patterns into Sub-patterns to be Stored into TCAMs

For the time being, let us assume that we have identified a good TCAM width *w* for the given signature set. For example, Table 5-2 shows a pattern set with four patterns and the TCAM width is set to be four bytes. Long patterns (Patterns 1-3) are divided into shorter patterns. We call the first *w* bytes *prefix pattern*s and the remaining part *suffix pattern*s. Patterns shorter than *w* bytes (Patten 4) remain intact. The selection of *w* and the tradeoff between a single cycle "long" match and many cycles of "short" matches will be discussed in Section 5.7.

Table 5-2. A Long pattern example (TCAM width *w* = 4).

Pattern Index	Pattern Contents	Prefix Patterns	Suffix Patterns
1	ABCDABCD	ABCD	ABCD
2	DEFGABCDL	DEFG	ABCDL
3	DEFGDEF	DEFG	DEF
4	DEF	-	-

Prefix patterns can be fit into TCAM directly since they are *w* bytes. After matching a prefix pattern, one option is to use software to compare the suffix patterns. However, there may be multiple suffix patterns sharing the same prefix pattern as in patterns 2 and 3 in Table 5-2. In such cases, the computation costs for software comparisons are quite high.

Our approach places the suffix patterns into the same TCAM as the prefix patterns. If a suffix pattern is longer than *w* bytes, we need to divide it into multiple sub-patterns of *w* bytes each. The suffix patterns can be less than *w* bytes, or not exactly a multiple of *w* bytes. Hence, they may generate very short sub-patterns. For example, a pattern "ABCDE" generates a short sub-pattern "E" for *w* = 4. We can pad these short patterns with 'do not care' states to make them *w* bytes. The problem with this approach is that small patterns that are only one or two bytes greatly increase the probability of TCAM hits. Each TCAM hit demands extra processing to check whether the match is a valid pattern or if combining it with any previously matched pattern constitutes a long pattern match. Hence, it is computationally inefficient to divide long patterns into very short suffix patterns that generate many TCAM hits.

To overcome this problem, we pad the short suffix pattern in the front by the tail of previous pattern. For example, the suffix pattern of "DEFGDEF" is "DEF". We pad it with the tail of previous prefix pattern ("G") to make it exactly *w* bytes ("GDEF"). Another example is "DEFGABCDL", the suffix pattern "ABCDL" is divided into two suffix patterns of *w* bytes each: "ABCD" and "BCDL".

Table 5-3. Patterns in the TCAM.

TCAM Index	Content
1	ABCD
2	DEFG
3	BCDL
4	GDEF
5	DEF?

We can order the unique prefix patterns and suffix patterns in any order because they all have the length *w*, so none of them covers another unless they are identical. For patterns that are naturally shorter than *w* bytes (e.g., pattern 4 in Table 5-2), as before, we pad them with 'do not care' at the end and sort them according to the descending order of lengths. Table 5-3 shows the TCAM layout for patterns in Table 5-2.

Now, we have broken long patterns into sub-patterns and stored them in the TCAM. When we perform a lookup in the TCAM, it reports matching of these sub-patterns. For these sub-pattern matching results, we need to record them in the memory so that we can correlate them with the future suffix pattern matching to identify long patterns. In the next subsection, we introduce several in-memory data structures for helping us correlate the sub-pattern matching results.

5.5.2 Data Structures in Memory

As in shown in Figure 5-3, in addition to the patterns stored in the TCAM, there are three data structures to be stored in memory (e.g., SRAM) for matching long patterns: the *pattern table*, the *matching table* and the *partial hit list.* We illustrate the detail information of these tables in Figure 5-4. The pattern table and the matching table are pre-computed based on pattern information and partial hit list is generated at runtime. The pattern table stores the information for each sub-pattern. It helps to identify whether the *current* matched pattern is a short pattern, a prefix pattern or a suffix pattern. The partial hit list records the *previously*

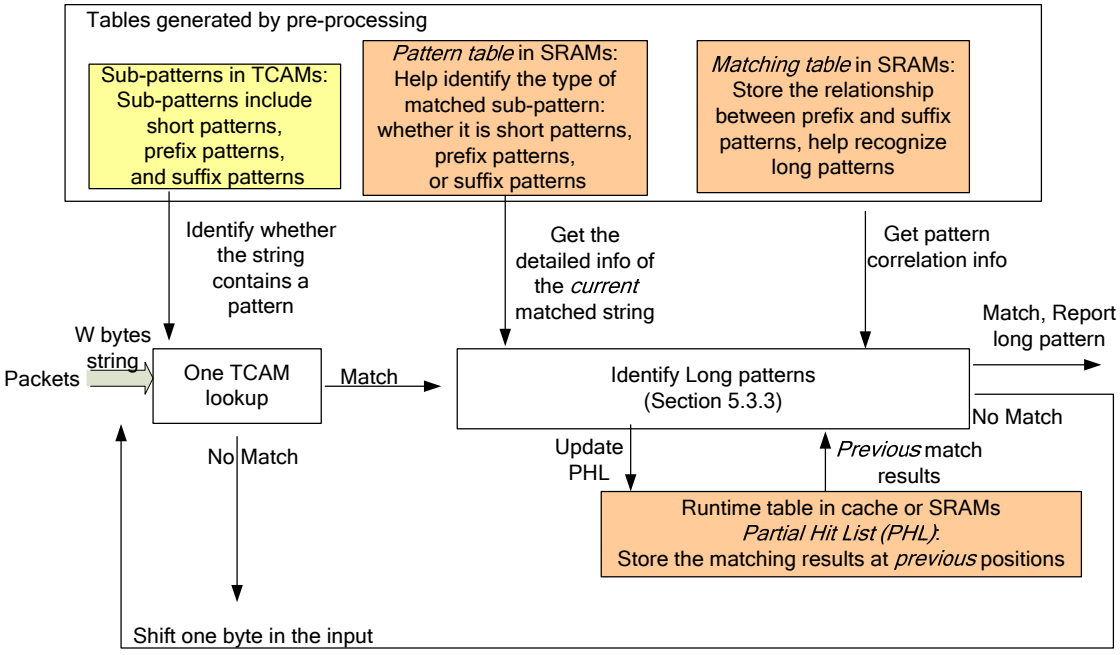


Figure 5-4. Tables used in our scheme.

matched sub-patterns. The matching table records the correlations between sub-patterns. Using these three tables, we can correlate the current match pattern with the previously matched patterns to identify long patterns. Next, we explain these tables in detail.

Pattern Table

All the patterns (simple, prefix, and suffix patterns) are put into a single TCAM. When matching an entry in TCAM, we need to check whether it is a short pattern, a prefix pattern, or a suffix pattern. The pattern table (Table 5-4) records such information. Each line in the table corresponds to one TCAM entry.

The second column records whether it is a matching of a short pattern. For example, from a hit of "DEFG", we can infer a matching of "DEF". The third column shows the prefix pattern information. A positive number illustrates a valid pattern while "-1" indicates otherwise. Since not every entry in TCAM is related to a prefix pattern, we number the valid prefix patterns and refer to an entry in the resulting enumeration by a *prefix pattern index*.

Table 5-4. Pattern table.

Index (Content)	Simple Pattern Index	Prefix Index	Suffix Index
1(ABCD)	-1	1	1
2(DEFG)	4(DEF)	2	-1
3(BCDL)	-1	-1	2
4(GDEF)	-1	-1	3
5(DEF ?)	4(DEF)	-1	-1

Table 5-5. Partial hit list.

Index	Position
1	1

Column four stores the suffix pattern index. Note that this index is separately built and thus is independent of the prefix pattern index.

Now, we can identify sub-patterns through a TCAM lookup and then use the matched index to perform one SRAM lookup into the pattern table to get the information of these sub-patterns. Next we need to store the matched prefix patterns in a table called partial hit list in memory so that later we can correlate them with suffix patterns to identify long patterns.

Partial Hit List (PHL)

The Partial Hit List (PHL) is used to record the matched pattern results as shown in Table 5-5. When matching a prefix pattern, record it in the PHL. For example, when matching pattern "ABCD" at the first four bytes of the packet, we record the index (*1* in this example) and the starting position of the pattern in the packet (1st byte in this example) in the PHL. Later, when we match its matching suffix, e.g., "ABCD", we can retrieve the previous matching result of "ABCD" and concatenate these two patterns into a long pattern "ABCDABCD". Not all prefix patterns and suffix patterns can be put together to generate long patterns. Hence, we also need a table to record the valid combination of suffix and prefix patterns. In the

Table 5-6. Matching table.

Prefix Index	Suffix Index	Distance	Matched Long Pattern Index
1(ABCD)	1(ABCD)	4	1(ABCDABCD)
2(DEFG)	1(ABCD)	4	3*(DEGFABCD)
2(DEFG)	3(GDEF)	3	3(DEGFDEF)
3(DEGFABCD)	1(ABCD)	4	1(ABCDABCD)
3(DEGFABCD)	2(BCDL)	1	2(DEFDABCDL)

following, we explain how the matching table stores the correlation information between patterns.

Matching Table

After identifying the prefix and suffix patterns, we assemble them to recover long patterns. A matching table (Table 5-6) stores all the valid combinations. For example, the combination of the prefix pattern "DEFG" and the suffix pattern "ABCD" yields a new prefix pattern (annotated by 3*). We give the new prefix pattern ("DEFGABCD") a value of 3 as the prefix pattern index and store it back to the PHL. Later, when we match suffix pattern "BCDL" at the next position, we can lookup the matching table and conclude that we have matched pattern "DEGFHIJKL".

At first glance, looking up entries in the mapping table appears to be a slow process, since we need to search through the mapping table to check whether one combination is valid. However, in a real system, we can trade space for speed. The first three columns (prefix index, suffix index, and distance between patterns) can be used as indices of a three-dimensional array and we do not need to store those columns. Only long patterns matched at the corresponding indexed space are stored and 'no match' (-1) are placed at all the other invalid combination places. In this manner, we can decide whether a combination is valid or

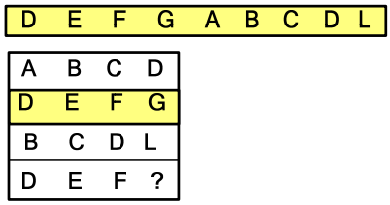
not with only one memory lookup. The total memory consumption is *a*b*w*, where *a* is the prefix index size, *b* is the suffix pattern index size, and *w* is the TCAM width.

5.5.3 Algorithms for Long Patterns

As described in Section 5.5.1, long patterns are handled by dividing them into sub-patterns which are then stored in TCAM. This section presents an algorithm for identifying long patterns, by linking the matched sub-patterns together to identify long patterns.

Our approach for identifying a long pattern is as follows. Initially we pre-process the patterns and generate the pattern table and mapping table using the methods we previously described in Section 5.5.2. The PHL is initially set to empty. Given an input, we take the first *w*-byte (byte 1 through *w*) string and perform a TCAM lookup to check whether this string matches any of the sub-patterns. If there is no match, we shift forward one byte and perform a TCAM lookup at the next position (bytes 2 through *w*+1) in the packet. If we find a TCAM match at a given position, we consult the pattern table to check whether it is a short pattern, and/or prefix pattern, and/or suffix pattern. Note that string can be a short, prefix and suffix pattern at the same time. For example, in the string "DEFG" in the example in Table 5-4 is a prefix pattern, but it also contains a short pattern "DEF". After identifying the types of the string, we perform one or several of the following three actions.

- *Action 1 (short pattern case)*: if the matched pattern implies a short pattern, report the matching result immediately.
- *Action 2 (suffix pattern case)*: we check whether the combination of this pattern with any prefix pattern in PHL forms a valid long pattern through consulting the mapping table.
- *Action 3 (prefix pattern case)*: we insert the pattern into the PHL if it was not previously inserted by Action 2.



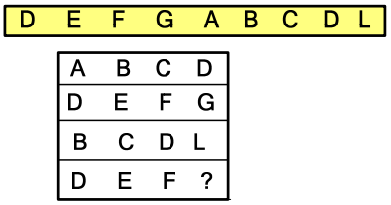
Partial Hit List (PHL) after this position

Position	Sub-pattern Index
1	2 (“DEFG”)

Position 1: match “DEFG”.

“DEFG” implies a short pattern “DEF”.

“DEFG” is also a prefix pattern with index 2, insert such information to the PHL.

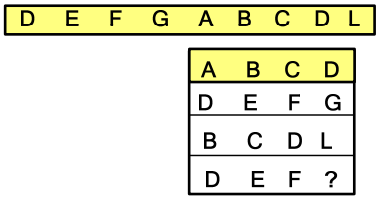


PHL after this position

Position	Sub-pattern Index
1	2 (“DEFG”)

Position 2: no match.

No match for position 3 and 4 either. These two positions are omitted in the figure.



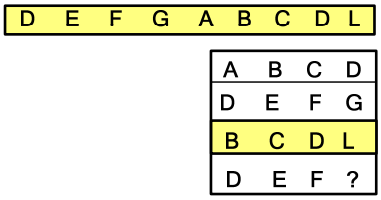
PHL after this position

Position	Sub-pattern Index
5	3 (“DEFGABCD”)

Position 5: match “ABCD”

“ABCD” is a suffix pattern, combined with prefix pattern 2 (“DEFG”) in the PHL yields another prefix pattern “DEFGABCD” (index in the mapping table is 3). The old record in PHL can now be deleted since it is *w* position away from the next position.

“ABCD” is prefix pattern, but it can be implied through “DEFGABCD”, so won’t insert it into the PHL again.



PHL after this position

Position	Sub-pattern Index

Position 6: match “BCDL”.

“BCDL” is a suffix pattern. Combining with “DEFGABCD” in the PHL, report a long pattern “DEGFABCDL”.

Figure 5-5. Matching process for input “DEGFABCDL”

Note that suffix patterns must immediately follow the prefix patterns to form long patterns, so we can delete prefix patterns that are more than *w* bytes. In addition, at each position, we only insert one item into the PHL (according to the restriction in Action 3). Therefore, the size of PHL is bounded by the TCAM width *w*.

We use an example (Figure 5-5) to illustrate our pattern matching algorithm for long patterns. Suppose the input packet is "DEFGABCDL" and we want to search for the patterns shown in Table 5-2. First we divide each of the patterns into sub-patterns and insert them into the TCAM (Table 5-3). We then pre-compute the pattern table (Table 5-4) and matching table (Table 5-6) using the scheme we proposed in the previous section. These tables remain static during the pattern matching process. The only table that changes during the matching process is the partial hit list (PHL). Initially, it is set to be empty.

At the first position, we match "DEFG". Through consulting the pattern table, we know that "DEFG" is a short pattern and also a prefix pattern. Hence, we perform Actions 1 and 3. "DEFG" implies a match of pattern "DEF", so we report a matching of "DEF" right away according to Action 1. This string is again a prefix pattern, so we store such information in the PHL according to Action 3. We've finished the two actions for this position. Next we shift one character to position 2 and lookup "EFGA" into the TCAM. Here the lookup will not result in a match, so we move on to the next position in the pattern. Similarly, we find no match for position 3 and 4.

When we move to the 5^{th} position, we get a match on the pattern "ABCD". "ABCD" is both a suffix pattern and a prefix pattern, so we perform Actions 2 and 3. According to Action 2, we try the combination of this pattern and the prefix pattern in the PHL ("DEFG"). Through consulting the mapping table, we get another prefix pattern "DEFGABCD". We store this prefix pattern back to the PHL. Since we already inserted one item into the PHL,

we do not need to insert “ABCD” as a prefix pattern again according to Action 3. This is because “DEGFABCD” is already in PHL that implies “ABCD”.

Finally, we move to the 6th position and match “BCDL”. “BCDL” is a suffix pattern so we only need to perform Action 3. Upon combining it with the prefix pattern “DEFGABCD” in PHL, we identify the long pattern “DEFGABCDL”.

5.6 Composite Patterns

The methods presented in the previous two sections are for fixed strings. They are not sufficient for intrusion detection systems such as SNORT, which require many composite patterns. In this section, we extend our algorithm to handle these composite patterns. According to the definition in Section 5.3, there are two types of composite patterns, correlated patterns and patterns with negations. In this section, we explain these two cases. At the end of the section, we propose a method to support case-insensitive patterns.

5.6.1 Correlated Patterns

Correlated patterns denote a series of patterns, i.e., patterns followed by other patterns like “ABCD” followed by the pattern “DEFG” within 4 bytes from the end of the first pattern. We call the units of the pattern series sub-patterns (in this example, “ABCD” and “DEFG”). Matching a correlated pattern is similar to a long pattern: a long pattern is just several sub-patterns and the distance of these patterns must be exactly *w*. Hence, the scheme for long patterns can be extended to correlated patterns. The matched sub-patterns are also recorded in the PHL. The only difference between matching correlated and long patterns is that the partial hit record for sub-patterns cannot be removed after *w* positions because the distance between two sub-patterns can be larger than *w*.

5.6.2 Patterns with Negations

The *negation (!)* of a pattern refers to detecting the absence of the pattern from the input. Negations typically occur in conjunction with another pattern and when they do they are usually the second pattern in the sequence. For example, *content : "USER" ; nocase ; content : !"|0a|" ; within: 50.* This pattern corresponds to an observation of the pattern “USER” and the absence of the pattern "|0a|" (return) within the next 50 bytes.

The identification of a pattern’s negation is similar to the identification of a correlated pattern. We first ignore the negation (!) and put the second pattern (|0a| in our example) into the TCAM. After matching the first sub-pattern “USER”, we record this matching information in the PHL. In the next 50 bytes, we then inspect whether there is a hit for "|0a|". The difference between matching a negation pattern and a correlated pattern is that the action performed upon observation of the second pattern is the opposite. If we find the second pattern (|0a|), this means we did not match the first pattern and so we can now remove the index for “USER” from the PHL. Otherwise, after 50 bytes with no matching of the second pattern, we report a hit of pattern "USER" and !"|0a|".

5.6.3 Patterns with Wildcards

Some signature databases specify patterns with the “no case” keyword. This means that either an upper case or a lower case of the pattern is considered valid. In ASCII, the numeric difference between a lower case character and its corresponding upper case character is 32. Since this happens to be a power of 2, TCAMs can support case-insensitive matching easily with a ‘do not care’ bit. For example, the ASCII code for letter “A” is 65 (binary form 0100 0001) and letter “a” is 97 (binary form 0110 0001). So we can represent case insensitive letter *a* in the TCAM as (01?0 0001). If there is a requirement for a fixed width wildcard for

any characters, then we can just put ‘do not care’ states in all their corresponding positions in the TCAM.

5.7 Analysis of the Scheme

In the previous three sections, we have proposed fast matching methods for fixed string patterns, long patterns, and composite patterns. In this section, we analyze the performance of the proposed schemes using two metrics.

The first metric is *memory consumption*. We want to minimize usage of SRAM, and more importantly TCAM, as TCAM is comparatively expensive with current manufacturing technologies.

The second metric is *pattern scanning rate* as pattern matching schemes must sustain the line rates. Memory lookups are usually the bottleneck of packet processing systems [88]. Hence, the pattern scanning rate is mostly controlled by the memory access speed and the number of memory lookup. For each byte position, our scheme needs to lookup TCAM once. Besides this fixed overhead, the number of extra memory lookups is affected by two factors. First is the number of TCAM hits since each TCAM hit requires one memory lookup in the pattern table. Second is the size of the PHL because we need to access the mapping table once for each item in the PHL for a matched suffix pattern.

As the TCAM width is configurable, in Section 5.7.1, we study the effect of changing TCAM width on the memory consumption and the number of memory lookups. In Section 5.7.2, we explain the memory access process and study the effect of memory access rate on the overall scanning rate. As our pattern matching schemes are designed for packet payload scanning applications, our scheme as described thus far is subject to malicious attack. Finally, in Section 5.7.3, we explain the methods for detecting malicious packets that are aimed at slowing down the system. Note that in this section, we make an assumption that patterns are

independent of each other for ease of analysis. This assumption may not be true in real pattern sets. We will present simulations results using the real pattern set in Section 5.8 and show that our analyses mostly agree with the real life cases.

5.7.1 Analysis Considering the TCAM Width

As we mentioned in Section 5.2, the width (w) of TCAM is configurable. In this section, we study the impact of TCAM width on matching long patterns. We will perform analysis of correlated patterns in Section 5.7.3. In this section, we first study the effects of TCAM width on memory usage, including the TCAM space usage and SRAM space usage. Then we study how modifying these parameters affects the number of memory accesses, which in turn determines the pattern matching rate. As explained previously, the number of memory accesses is affected by two factors: the number of TCAM hits and the size of partial hit list. Therefore, we explain the effect of TCAM width on these two factors later in this section.

Effects on TCAM Space

Suppose we have a total of k patterns, each with m_i bytes. If we set the TCAM width to w, then each pattern will be divided into $\lceil m_i / w \rceil$ prefix or suffix patterns, where $\lceil\ \rceil$ denotes rounding up. As each TCAM entry is w bytes and $\lceil m_i / w \rceil$ entries are needed, the total TCAM space required to accommodate these patterns is $w * \sum \lceil m_i / w \rceil$. It increases as w increases because short patterns and suffix patterns need to be padded (the padding effect is represented by $\lceil\ \rceil$ in the equation) to the TCAM width.

Effects on SRAM Memory Space for Mapping Table

The mapping table is a three dimensional table that can be accessed with the prefix index, suffix index, and the distance between them. The size of the prefix pattern index is $\sum_i (\lceil m_i / w \rceil - 1)$, which can be proved through two cases. Case 1: If a pattern is shorter than w

bytes, it has no prefix pattern. This agrees with $\lceil m_i / w \rceil - 1 = 0$. Case 2: For a pattern longer than *w,* although there is one prefix pattern, it generates new prefix patterns after intermediate suffix patterns are matched. Hence, it also requires $\lceil m_i / w \rceil - 1$ indices. Therefore, independent of the pattern length, the number of prefix indices is $\lceil m_i / w \rceil - 1$ per pattern and the total is $\sum_i (\lceil m_i / w \rceil - 1)$ for all patterns. Similarly, the size of the suffix pattern indices is also $\sum_i (\lceil m_i / w \rceil - 1)$. For long patterns, the maximum distance between a prefix pattern and a suffix pattern is *w*. The mapping table size is the product of the above three dimensions, which are $\sum_i (\lceil m_i / w \rceil - 1)$, $\sum_i (\lceil m_i / w \rceil - 1)$, and *w* respectively. Multiplying them together, we get $w * (\sum_i (\lceil m_i / w \rceil - 1))^2 = O(1/w)$. Therefore, the size of mapping table decreases as *w* increases.

We have shown that the TCAM width greatly affects TCAM and SRAM memory usage. In summary, the TCAM space increases as the TCAM width increases, while the SRAM memory decreases when the TCAM width increases. Next, we check the effect of the TCAM width on the number of memory lookups. As mentioned previously, besides the fixed overhead – one TCAM lookup per byte position, the number of memory lookups is affected by two factors: the probability of TCAM hits, and the size of the partial hit list. Next, we study these two factors in detail.

Probability of TCAM Hits

We distinguish between two types of TCAM hits. The first is a *real hit*, which reports a pattern match immediately. For a pattern longer than *w*, we define the matching of the last suffix pattern as a real hit. As at least one real hit is required for one matched pattern, the number of real hits is the lower bound of the number of TCAM hits for identifying all the patterns. The other type of hit is an *associate hit*—an intermediate hit that may lead to a real hit, i.e., each prefix pattern hit is an associate hit. Associate hits incur extra computation, so

we want to minimize the probability of associate hits. Assuming packet contents are random, for any *w* bytes in the packet, there are $(2^8)^w$ possible values and the probability of matching one particular pattern of length *w* is $1/(2^8)^w$. As we have analyzed before, there are $\sum_i (\lceil m_i / w \rceil - 1)$ prefix patterns of *w* bytes each. If we assume patterns are independent, the probability of having an associated hit at each position is $\frac{\sum_i (\lceil m_i / w \rceil - 1)}{(2^8)^w}$, which decreases dramatically when *w* increases. For example, suppose we have 2000 patterns of 200 bytes each. Setting *w* to be 4 bytes, the associate hit rate at each position is 2.2e-5. If w is 8, it is 2.6e-15, which is very low. Therefore, to decrease the extra cost for unnecessary TCAM hits, a large *w* is preferred.

Effects on PHL size

As discussed in Section 5.5.3, PHL has an upper bound of *w* for pattern sets that contain long patterns only. Let's study the probability of two neighboring overlapping TCAM hits. Suppose we observe a match at position *i*. Such an observation imposes extra constraints on whether a match could occur at position *i+j (j<w)* because the last *w-j* bytes of the first position must be identical to the first *w-j* bytes of position *i+j*. Assume that patterns are independent, the number of overlapped pattern pairs is small and hence *PHL* is small. If we relax this constraint and assume these positions are independent, the expected prefix pattern list size is *w* times the independent TCAM associate hit rate ($\frac{\sum_i (\lceil m_i / w \rceil - 1)}{(2^8)^w}$ as we showed previously). This comes to $w * \frac{\sum_i (\lceil m_i / w \rceil - 1)}{(2^8)^w}$, which approaches zero quickly as *w* grows. For example, suppose there are 2000 patterns with 200 bytes each. If *w* is set to be 4 bytes, the expected size of PHL at each position is 8.8e-5. If w is 8, it is 2e-14, which is well below 1. We will analyze the PHL for correlated patterns in the Section 5.7.3.

Summary on the impacts of TCAM width

The above analyses show that a small value of w can save TCAM space. However, a small w also generates many prefix and suffix patterns, which results in a large mapping table. In addition, since each entry in the TCAM is small, a small w reports many matches, creates a large PHL and requires many matching table lookups. Therefore, if there is enough TCAM space, we should set w larger than most of the pattern sizes and allow only a very small number of patterns to be divided into prefix and suffix patterns.

5.7.2 Analysis of Memory Lookups

Memory lookups are usually the bottleneck of packet processing systems [88]. There are two types of memory lookups in our scheme: TCAM lookups and regular memory (e.g., SRAM) lookups in the pattern table and matching table. TCAM lookups and memory lookups can be pipelined as illustrated in Figure 5-6. We can perform memory lookups for a current position while consulting TCAM with the data at next position. Suppose we have a packet of n bytes and the TCAM lookup time is a for each lookup. We will have a deterministic TCAM processing time of $n*a$.

If the TCAM reports a miss, no extra memory lookup will be initiated in this position i and the memory lookup process is idle. Otherwise, the proposed scheme will first perform one memory lookup in the pattern table. If the matched pattern is a valid suffix pattern and there are j_i items in the current PHL, we need another j_i memory lookups in the matching table. Hence, a maximum of j_i+1 memory lookups will be issued for a TCAM hit. The memory lookup time may be shorter or longer than the TCAM lookup time, thus the memory lookup process may be backlogged. For example, in Figure 5-6, positions 1, 2, and 3 all have

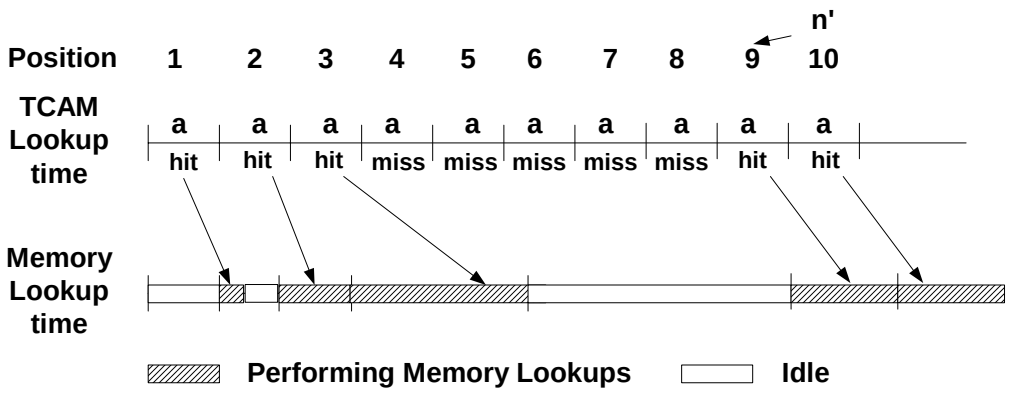


Figure 5-6. Memory lookup process.

TCAM hits, increasing the amount of time spent processing memory lookups. Later when there are some TCAM misses, the memory lookup process can catch up with the TCAM lookup speed. Therefore, only the last memory backlog position (*n'*) is important. The overall packet scan time is the sum of the time needed for the TCAM accesses up to this position (*n'*a*) and the memory lookup time after this position ($\sum_{i=n'}^{n}(j_i+1)*h_i$). Here h_i is the TCAM hit rate. j_i is usually a very small number (<1) and the TCAM hit rate (h_i) is also low as we analyzed in Section 5.7.1. Therefore, the second term $\sum_{i=n'}^{n}(j_i+1)*h_i$ is small. In such a case, the speed of the pattern matching is dominated by the TCAM lookup time. Assuming TCAM lookup time is *4* ns, the total time to scan an *n* bytes packet is *4n* ns. This yields a matching speed of *8*n/4n = 2* Gbps if we have a small TCAM hit rate and PHL size. We will verify these analyses through simulation later in Section 5.8.3.

Having explained the memory access process (which affects the packet scanning rate), next we study the policies for protecting the system against malicious attack. We study this because high speed processing is an essential requirement for packet processing systems. Many existing systems process normal traffic at high speeds, but perform slowly for certain rarely-occurring packets. Hence, without protection techniques, intruders could flood the system with these hard-to-process packets to substantially reduce system performance.

5.7.3 Protection against Malicious Attacks

One application of our proposed pattern matching scheme is network intrusion detection systems. Such a system itself can be exposed to malicious attacks. For example, packet generators like SNOT [89] generate packets that demand a lot of processing power to slow down the intrusion detection systems. We need to protect our pattern matching scheme against such attacks.

Our pattern matching scheme for the correlated patterns is subject to such attacks. Unlike long patterns, where we can discard the partial hit results that are w positions away, for correlated patterns, the distance between two sub-patterns can be larger than w. In addition, each sub-pattern can be smaller than w bytes, which generates a higher TCAM hit rate than longer patterns. Intruders may intentionally send packets that cause a lot of partial hits for the correlated pattern to create a long PHL. Later, when a suffix pattern is matched, a large number of memory lookups have to be issued and the system performance degrades dramatically.

To deal with this kind of attack, we study the size of PHL for correlated patterns. First we answer the question: if we match a pattern of length m at position i, what is the probability that we can construct an input to match another pattern at position $i+j$? If j is one, this means matching two overlapping patterns that are one byte apart. Such probability is low because it requires the first $m-1$ bytes of the second pattern are the same as the last $m-1$ bytes of the first pattern. Given k independent patterns, the probability of two overlapping patterns that are one byte apart is $1- ((2^8)^{m-1})!/(((2^8)^{m-1}-k)!*((2^8)^{m-1})^k)$. For example, $k = 1000$ and $m = 4$, it is 0.029, which is low.

When $j = m$ and the two patterns do not overlap, intruders can pack the sub-patterns consecutively to form an n byte packet. This packet generates matches at every n/m positions,

where *m* is the shortest sub-pattern length. Thus the PHL can have *n/m* items or more. To limit the PHL size, we recommend limiting the maximum distance between two sub-patterns to be considered as correlated. This recommendation is reasonable because in practice, patterns very far apart are unlikely to be considered correlated. For example, the maximum distance between patterns in SNORT patterns is 255.

Now, we have finished analyzing our proposed scheme's worst-case performance using theoretical techniques. This is based on the assumption that patterns are independent. Next we apply our scheme to the real world pattern sets and traces to demonstrate the effectiveness of the proposed scheme.

5.8 Simulation Results

In this section, we test the effectiveness of our proposed TCAM-based pattern matching scheme. We use two pattern sets. The first is a virus signature set from ClamAV [46], which contains simple patterns only. The second is from the SNORT intrusion detection [10] system with many correlated patterns. We start this section with a methodology section in 5.8.1. We present simulation results on the ClamAV pattern sets in Section 5.8.2 and SNORT patter sets in Section 5.8.3.

5.8.1 Methodology

We test our scheme on two complex pattern sets: ClamAV [46] and SNORT. ClamAV (Version 0.15) contains 1768 simple fixed string patterns. The SNORT system (v2.1.2) has 1836 string patterns. 1039 of them are simple patterns and 527 are correlated patterns with up to seven sub-patterns in one correlated pattern.

Two sets of real-world packet traces are used during the experiments. The first set is the intrusion detection evaluation data set from the MIT DARPA project [22]. It contains more

than a million packets. The second data set is from a local LAN with 20 machines at the UC Berkeley networking group, which contains more than six million packets. The characteristics of the MIT dump are very different from Berkeley dump. The former consists mostly of long packets containing some worms (but not viruses), with the average packet payload length 507 bytes. In the Berkeley dump, however, most packets are normal traffic without worms and viruses, with 68 bytes on average in the packet payload. A large fraction of packets in the Berkeley trace are ICMP and ARP packets that have very short packet payloads, which makes detection very easy because not many patterns can occur in such a small payload.

Although the MIT dump contains intrusions, it does not contain many computer viruses. To test the performance of our scheme on ClamAV virus database under malicious input, we generate synthetic traffic containing viruses to stress test our scheme. We generated four sets of synthesized data, each with 1, 10, and 100 randomly inserted virus patterns per testing packet respectively.

For all the test traces we record the average and maximum Partial Hit List (PHL) size for each packet. As we mentioned in Section 5.7, the size of PHL denotes the number of memory lookups needed per byte position. Hence, the size of PHL transitively affects the scanning rate. We used three metrics for the PHL size over the entire trace data. *Avg* is the mean of the average PHL size over all packets. *AvgMax* denotes the mean of the maximum PHL sizes over different packets. *Max* records the maximum size over all packets, which denotes the maximum number of entries in memory during the run.

5.8.2 Results on ClamAV Pattern Set

ClamAV (Version 0.15) has 1768 simple patterns. The average pattern length is 55 bytes. Figure 5-7 plots the distribution of the pattern length. It varies from 6 bytes to 2189 bytes.

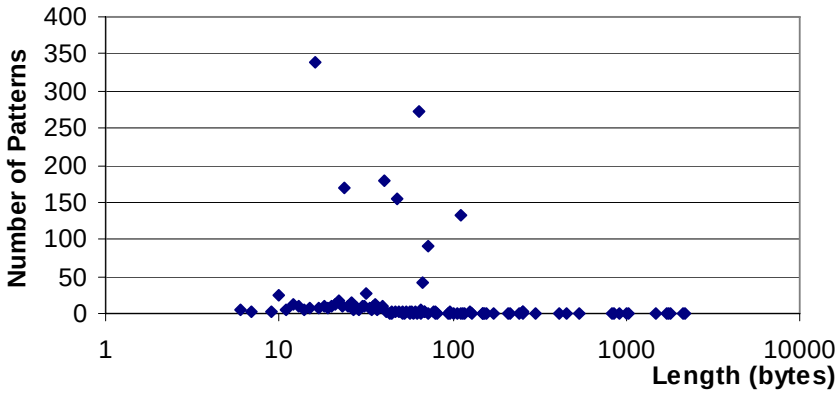


Figure 5-7. Distribution of pattern length over the ClamAV rule set.

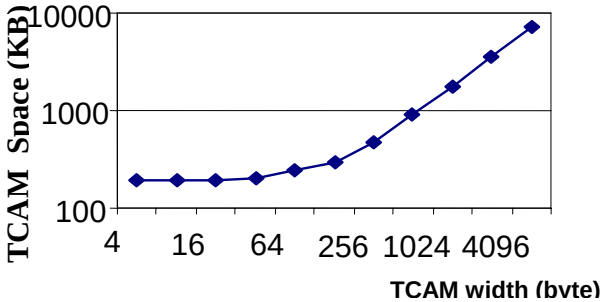


Figure 5-8.TCAM space consumed vs. TCAM width.

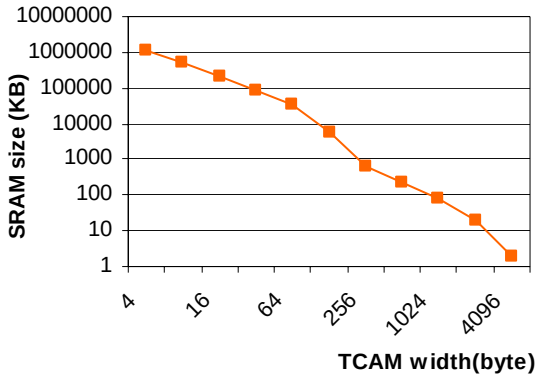


Figure 5-9. Mapping table size vs. TCAM width.

With such a large variance in pattern lengths, selection of the TCAM width *w* is critical. Figure 5-8 shows the total TCAM space needed to accommodate all the patterns under different values of *w*. When *w* increases, the TCAM space requirement increases too due to the padding for short and suffix patterns. This agrees with our analysis in Section 5.7.1. The

Table 5-7. PHL size for ClamAV signature set.

TCAM Width	MIT Dump			Berkeley Dump		
	Avg	***AvgMax***	***Max***	***Avg***	***AvgMax***	***Max***
4	0.042	0.27	4	0.03	0.48	4
8	4.8e-6	5.6e-4	8	1.e-6	1.9e-5	7
16	0	0	0	4.3e-7	5.8e-6	3
32	0	0	0	0	0	0
64	0	0	0	0	0	0
128	0	0	0	0	0	0

size of the mapping table in SRAM, however, is negatively correlated to the TCAM width as shown in Figure 5-9. When w is small, a long pattern generates many prefix and suffix patterns. Therefore, the number of prefix and suffix indices grows, resulting in a large mapping table. Combining the analyses from Figure 5-8 and Figure 5-9, we choose w to be 128 bytes and use a 240 KB TCAM and 657 KB SRAM.

5.6.2.1 Test Results on Real Data

Table 5-7 shows the PHL size for both the MIT and Berkeley traces. Since these two traces do not contain many viruses, the PHL size is extremely low when the window size is reasonably large. When the size of PHL is small, the memory lookup process is mostly idle and the system performance is bounded by the TCAM access rate only. So, we can achieve 2Gbps rate with a 240 KB TCAM.

5.6.2.2 Test on Synthesized "Worst-case" Packets

We create synthesized data that contains a large number of virus signatures to stress test our scheme. We generate three sets of data, the first set contains a single virus patterns per packet, the second set contains ten patterns in a sequence per packet, and the last set contains one hundred patterns per packet. For these synthesized packets, many TCAM hits are generated, resulting in a larger PHL size compared to the real traffic. Figure 5-10 shows the average PHL size. It decreases quickly as we increase the TCAM width w. In addition,

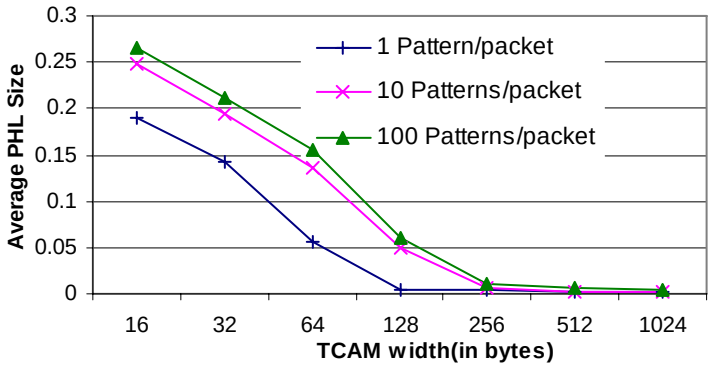


Figure 5-10. Average of PHL size over synthesized data.

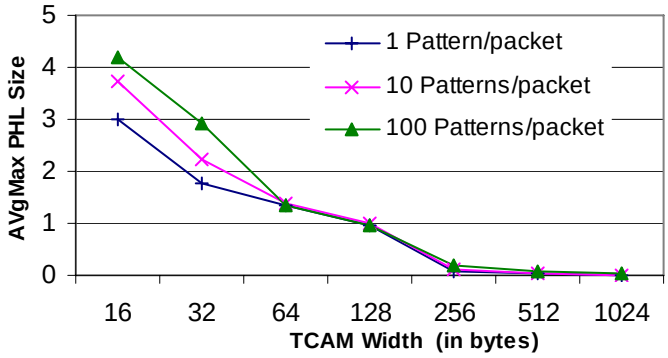


Figure 5-11. Average of Max PHL size per packet.

having multiple patterns in the packets does not increase the PHL size dramatically. This is because we can delete the prefix patterns that are *w* bytes ahead, so the number of patterns in a *w* byte window may not increase proportionally to the increase of the total number of patterns per packet.

The *AvgMax* PHL size per packet is notably larger than *Avg* as plotted in Figure 5-11. This shows that some contents in the packets cause backlogs in the memory lookup process. The effect of the TCAM width again has significant impact—if we set *w* to be 128 bytes or longer, the *AvgMax* PHL size is around one per packet. This means that even with this "worst-case" data, the memory lookup process can still finish within one or two cycles after the TCAM lookup process finishes. This has the same effect as increasing the packet size by one or two bytes. Given the fact that IP packets typically consist of at least tens of bytes and

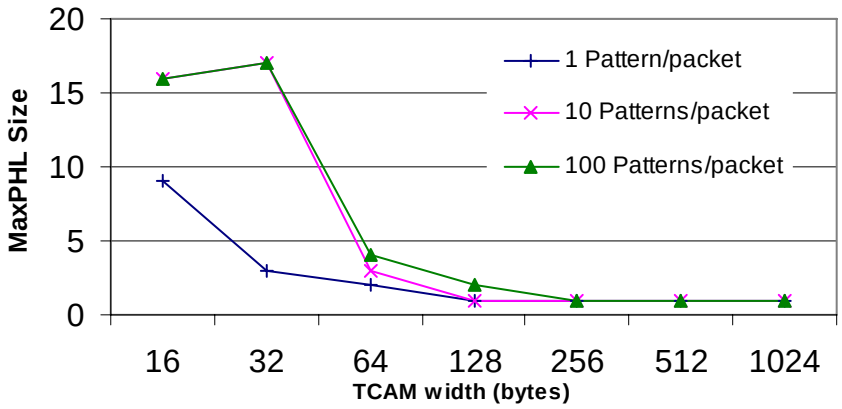


Figure 5-12. Max PHL size over all packets.

we do not need to perform pattern matching on the packet header, the impact of slightly increasing the "effective packet length" for matching purposes is negligible. Hence the packet scan rate is still 2Gbps over this set of synthesized data. Figure 5-12 illustrates the maximum PHL size over all packets. When *w* is small (e.g., 16), the maximum is small because the *max* PHL size is bounded by *w*. When *w* gets larger, the PHL size increases and then drops quickly because the probability of a TCAM hit becomes small for large *w*. There is a big difference in the PHL sizes between packets containing one virus pattern and ten patterns. However, as we increase the number of viruses per packet, the growth of max PHL size slows. This is because within *w* bytes, the number of viruses is limited, even though we can have a hundred viruses in the whole packet.

5.8.3 Results on SNORT Pattern Set

The current Version of SNORT (v2.1.2) contains 1836 string patterns. The lengths of patterns are much shorter than the ClamAV signature set: mostly from 10 bytes to 100 bytes. In addition, there is a noticeable amount of short patterns of one or four bytes. Among these string pattern patterns, 1039 are simple patterns and 527 are correlated patterns with up to

Table 5-8. PHL size for SNORT signature set.

Window Size	MIT Dump			Berkeley Dump		
	Avg	AvgMax	Max	Avg	AvgMax	Max
20	0.5523	2.7683	8	0.4702	1.5765	12
40	0.9881	3.5376	14	0.6500	1.8661	18
60	1.3151	3.9960	14	0.7313	1.9652	23
80	1.5491	4.2158	16	0.7587	2.0373	24
100	1.6867	4.3485	18	0.7661	2.0740	25
120	1.7725	4.4475	18	0.7669	2.0768	25
140	1.8308	4.5722	19	0.7669	2.0768	25
160	1.8800	4.6643	19	0.7669	2.0768	25
180	1.9244	4.7386	19	0.7669	2.0768	25
200	1.9662	4.8079	20	0.7669	2.0768	25

seven sub-patterns in one correlated pattern. We set the TCAM width to 128 bytes and the patterns can be mapped into a TCAM size of 295 KB.

Since SNORT has correlated patterns, we first test the impact of different window sizes ranging from 20 to 200 bytes. Compared to ClamAV, the PHL size is much larger as shown in Table 5-8. This is because the SNORT signature set contains a lot of short patterns. In addition, the size of PHL increases when the window size increases because it needs to keep the partial hit information longer. However, as the window size grows larger, PHL increases at a slower rate.

A large PHL is problematic since it requires many memory lookups and slows down the system. Therefore, we studied the total scanning time (including memory lookups and TCAM lookups). Since memory (e.g., SRAM) access is usually slightly faster than TCAM access rates, we simulated scenarios with different ratios of memory to TCAM access times, (which we call the *memory ratio*). For example, a value of one means memory access speed is equal to TCAM access speed, while 0.2 denotes that memory access speed is 5 times the TCAM access rate.

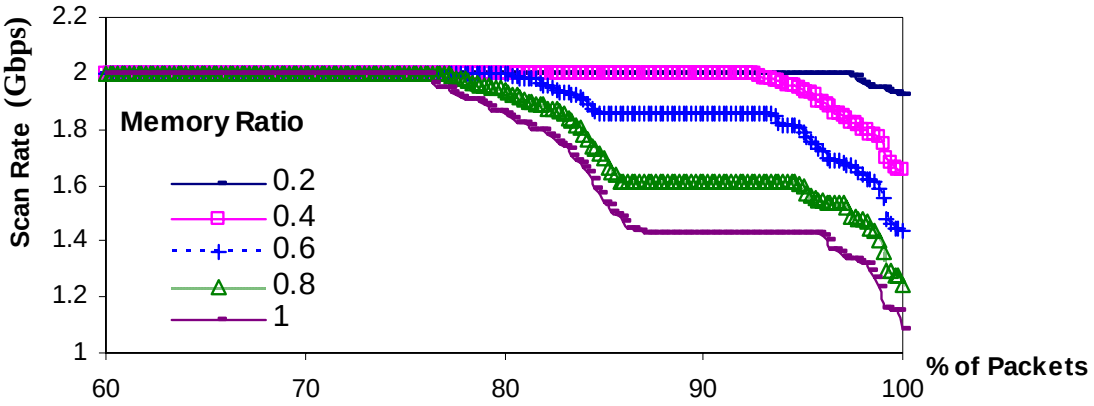


Figure 5-13. Effects of memory ratio on scan rates.

Figure 5-13 shows the impact of memory ratio on the scan rate, with each curve standing for one memory ratio setup. The y axis is the scanning rate, while the x axis is the percentage of packets that were processed at that rate. For example, a value of two (y axis) at 60 percent (x axis) means that 60% of the packets have a scan rate of 2 Gbps. Simulation results show that the scan rate is close to 2Gbps for most of the packets (80%) under all settings. Since the TCAM access rate is 2Gbps, the TCAM access speed is the bottleneck for these packets. For the remaining around 20% of packets, the memory access process is backlogged and therefore the overall system performance is lower than the TCAM access rate. Nevertheless, the scan rate is over 1Gbps for all memory ratios we tested. Sung, et al., reported that a 10 Gbps rate can be achieved by extending our scheme to jump multiple bytes at a time [90].

5.9 Conclusions

With the increasing importance of network protection from cyber-attacks, it is essential to develop mechanisms for building effective defenses against virus, worm, and denial of service attacks. The rapid rise in link bandwidths implies that network protection mechanisms must be capable of operating at multi-gigabit rates. A key operation for network

protection is pattern-matching to check for virus and worm signatures. In this chapter, we developed a TCAM-based scheme for matching fixed string patterns and composite patterns. Our proposed scheme can scan thousands of patterns simultaneously at gigabit rates. By evaluating its performance using multiple real-network traces we showed that it is indeed suitable for multi-gigabit operation. The scheme can also be extended to achieve even higher rates with larger TCAMs.

The methods presented in this chapter are only for fixed string and composite patterns. As mentioned in Section 5.3, there are many regular expression patterns in addition to these two patterns that are used in the network payload scanning applications. In the next chapter, we present schemes for fast regular expression matching.

6 Fast and Memory-Efficient Regular Expression Matching

Previous chapters provided schemes for high-speed, fixed-string matching. More recently, fixed string patterns are being replaced with the need to support regular expression pattern matching in packet scanning applications. This chapter explores algorithms for fast regular expression matching. We first show that memory requirements using the well-known traditional methods for this problem are prohibitively high for some patterns used in packet scanning applications. We then propose regular expression rewrite techniques that rewrite these patterns into memory efficient ones. In addition, we present grouping schemes that strategically compile a set of regular expressions into several scanners that run in parallel, resulting in a significant improvement of regular expression matching speed without much increase in memory consumption. We implement a fast and memory efficient regular expression scanner for real-world patterns of packet scanning applications. Our experimental results using real-world traffic collected from Berkeley and MIT show that our implementation achieves a factor of 12 to 42 speed improvement over a commonly used DFA-based scanner. Compared to the state-of-art NFA-based implementation, our DFA-based packet scanner achieves a 50 to 700 times speedup with just a 2.6 to 8.4 times increase in memory usage.

6.1 Introduction

As described in Chapter 2, packet content scanning (also known as Layer-7 filtering or payload scanning) is crucial to network security and network monitoring applications. The payload of packets in a traffic stream is matched against a given set of patterns to identify specific classes of applications, viruses, protocol definitions and so on.

Viruses and worms have been structured by their authors so as to defeat simple pattern matching approaches. For example, polymorphic worms and attacks to complex protocols make it impossible to enumerate all the possible signatures using explicit strings. Regular expressions, however, can express these patterns due to their rich expressive power. For example, a regular expression *"^Entry /file/[0-9.]{71,}//.*\x0aannotate\x0a"* is for detecting a Concurrent Versions System (CVS) revision overflow attack. This pattern searches for a fixed pattern "Entry/file/" followed by 71 or more 0 to 9 digits, then a fixed pattern "//" followed by some arbitrary characters (.*), finally the pattern "*\x0aannotate\x0a"*. Obviously, it is very hard to catch this type of attacks using fixed string patterns. As a result, regular expressions are replacing explicit string patterns as the pattern matching language of choice in packet scanning applications. In the Linux Application Protocol Classifier (L7-filter) [22], all protocol identifiers are expressed as regular expressions. Similarly, the SNORT [10] intrusion detection system has evolved from no regular expressions in its rule set in April 2003 (Version 2.0) to 1131 out of 4867 rules using regular expressions as of February 2006 (Version 2.4). Another intrusion detection system, Bro [91], also uses regular expressions as its pattern language.

As regular expressions gain wider adoption for packet content scanning, it is imperative that out matching algorithms maintain line-speed while using moderate amounts of memory. Unfortunately, these requirements are not met in many existing payload scanning

implementations. For example, when all 70 protocol filters are enabled in the Linux L7-filter [22], we found that the system throughput drops to less than 10Mbps on a 100Mbps network using a computer with 750 MHz PIII and 512 MB of memory, which does not satisfy the requirements for high speed edge networks, where gigabit Ethernets have become the norm. Moreover, we found that over 90% of the CPU time is spent in regular expression matching, leaving little time for other intrusion detection or monitoring functions. At the same time, the memory requirements using traditional methods are prohibitively high for many patterns used in packet scanning applications. Although many schemes (including our approach in Chapter 5) for fast string matching [85, 86, 92-97] have been developed recently in intrusion detection systems, they focus only on *explicit string patterns* and cannot be easily extended for fast *regular expression* matching.

In this chapter, we study the reasons that traditional regular expression techniques fail to provide high speed regular expression matching for networking applications, and then propose our scheme for constructing fast regular expression recognizers. The previous chapter already gave a brief review of the patterns used in packet scanning systems and states the general pattern processing goal. In this chapter, we focus on the regular expression patterns. We begin with a brief survey of traditional regular expression pattern matching methods in Section 6.2. In Section 6.3, we study the special characteristics of typical regular expression patterns used in network scanning applications. We show that some of these patterns lead to exponential memory usage or low matching speed when using the traditional methods. Based on these analyses, we propose two rewrite rules for specific regular expressions in Section 6.4. The rewrite rules can dramatically reduce the size of resulting DFAs, making them small enough to fit in memory. In Section 6.5, we develop techniques to intelligently combine multiple DFAs into a small number of groups to accelerate the matching speed, while avoiding the exponential growth in the number of states in memory.

Table 6-1. Features of Regular Expressions.

Syntax	Meaning	Example
^	Pattern to be matched at the start of the input	^AB means the input starts with AB. A pattern without '^', e.g., AB, can be matched anywhere in the input.
|	OR relationship	*A*|*B* denotes an occurrence of either *A* or *B*.
.	A single character wildcard	
?	A quantifier denoting one or less	*A*? denotes *A* or an empty string.
*	A quantifier denoting zero or more	*A** means an arbitrary number of *A*s.
[97]	Repeat	*A*{100} denotes a sequence of 100 *A*s.
[ ]	A class of characters	[*lwt*] denotes a letter *l*, *w*, or *t*.
[^]	Anything but	[^*n*] denotes any character except *n*.

Finally, we demonstrate the effectiveness of our rewriting and grouping algorithms through a detailed performance analysis using real-world payload scanning pattern sets in Section 6.6.

6.2 Taxonomy

In this section, we give a brief overview of regular expressions with a survey of the existing approaches for regular expression matching.

6.2.1 Introduction to Regular Expression

A regular expression describes a set of strings without explicitly enumerating them. Table 6-1 lists the regular expressions operators used to describe patterns within packets. An *anchor* ("^") enforces that a pattern must be matched at the beginning of the input. "|" denotes *or* relationship. "." is a single character wildcard. In other words, any character can be matched using ".". "?" is a quantifier representing zero or one and "*" denotes zero or more. "{}" can be used to define more specific occurrence restrictions. For example, "{3, 5}" stands for repeating three to five times. We can also use "[]" to define a class, i. e., characters inside the brackets form a class and they have *or* relationships. When "^" appears in "[]", it has a special meaning of exception. For example, *"[^\n]"* denotes anything but the return key.

Let us take the regular expression for detecting Yahoo messenger traffic as an example: "^(*ymsg*|*ypns*|*yhoo*).?.?.?.?.?.?.?[*lwt*].**xc0\x80*". Yahoo messenger now has more than 76 million users in April 2006 according to comScore Media Metrix [91] and it is significant source of network traffic. This particular pattern is included in the popular Linux L7-filter set [22] for detecting Yahoo messenger traffic. According to the analysis by Venkat [99], all the Yahoo messenger commands start with *ymsg*, *ypns* or *yhoo*. Therefore, this pattern first identifies "(*ymsg*|*ypns*|*yhoo*)". The next seven or fewer bytes contain command length and Version information that varies among packets. So this pattern ignores those by using ".?.?.?.?.?.?.?". Then it identifies a letter *l, w* or *t*. *l* stands for "Yahoo service verify", *w* denotes "encryption challenge command", and *t* represents "login command". This pattern ends with ASCII letters *c0* and *80* in the hexadecimal form because *0xC080* is the standard argument separator in hexadecimal notation.

6.2.2 Solution Space for Regular Expression Matching

Regular expression matching has been studied for decades. In this section, we quickly review the traditional approaches to the general regular expression recognition problem. At the end of this section, we point out typical patterns that are easy to match using these traditional methods as well as patterns that are relatively expensive (in terms of computation or memory consumption).

Finite automata are a natural formalism for regular expressions matching. There are two main categories: *Deterministic Finite Automaton* (DFA) and *Nondeterministic Finite Automaton* (NFA). This section provides a brief survey of existing methods using these two types of automata.

A DFA consists of a finite set of input symbols, denoted as Σ, a finite set of states, and a transition function δ [100]. In networking applications, Σ contains the 2^8 symbols from the

extended ASCII code. Among the states, there is a single start state q_0 and a set of accepting states. The transition function δ takes a state and an input symbol as arguments and returns a state. A key feature of DFA is that at any time there is only one active state in the DFA. An NFA is similar to a DFA except that the δ function maps from a state and a symbol to a set of new states. Therefore, multiple states can be active simultaneously in an NFA.

Using automata to recognize regular expressions introduces two types of complexity: automata storage and processing costs. A theoretical worst case study [100] shows that a single regular expression of length n can be expressed as an NFA with $O(n)$ states. When the NFA is converted into a DFA, it may generate $O(\Sigma^n)$ states, where Σ is the set of symbols. The processing complexity for each character in the input is $O(1)$ in a DFA, but is $O(n^2)$ for an NFA when all n states are active at the same time.

To handle m regular expressions, two choices are possible: processing them individually in m automata, or compiling them into a single automaton. The former is used in SNORT [2] and Linux L7-filter [1]. The latter is proposed in recent studies [101, 102] so that the single composite NFA can support shared matching of common prefixes of those expressions. Despite the demonstrated performance gains over using m separate NFAs, in practice this approach requires large numbers of active states. This has the same worst case complexity as the sum of m separate NFAs. Therefore, this approach on a serial processor can be slow. As given any input character, each active state must be serially examined to obtain new states.

In DFA-based systems, compiling m regular expressions into a composite DFA decreases the processing complexity over running m individual DFA. Specifically, a composite DFA reduces processing cost from $O(m)$ ($O(1)$ for each automaton) to $O(1)$, i.e., a single lookup to obtain the next state for any given character. However, the number of states in the composite automaton grows to $O(\Sigma^{mn})$ in the theoretical worst case. In fact, we will show in Section

Table 6-2. Worst case comparisons of DFA and NFA.

	One regular expression of length *n*		*m* regular expressions compiled together	
	Processing complexity	**Storage cost**	**Processing complexity**	**Storage cost**
NFA	$O(n^2)$	$O(n)$	$O(n^2m)$	$O(nm)$
DFA	$O(1)$	$O(\Sigma^n)$	$O(1)$	$O(\Sigma^{nm})$

6.5.2 that typical patterns in packet payload scanning applications indeed interact with each other and can cause the creation of an exponential number of states in the composite DFA.

There is a middle ground between DFA and NFA, called lazy DFA. Lazy DFAs are designed to reduce the memory consumption of conventional DFAs [24, 101]: a lazy DFA keeps a subset of the DFA that matches the most common strings in memory; for uncommon strings, it extends the subset from the corresponding NFA at runtime. As such, a lazy DFA is usually much smaller than the corresponding fully-compiled DFA and provides good performance for common input strings. The Bro intrusion detection system [3] adopts this approach. However, malicious senders can easily construct packets with uncommon strings to keep the system busy and slow down the matching process. As a result, the system will start dropping packets and malicious packets can sneak through.

Having explained the complexity of DFA-based approaches and NFA-based approaches, next we present some typical patterns that are easy, in terms of computation and memory requirement, to match and patterns that are hard to match.

Easy-to-match patterns

It is obvious that patterns with fixed strings only are very easy to match. Patterns starting with an anchor ("^"), known to start at the initial position, are relatively easier to match than those without. Many protocol patterns, such as the previously mentioned Yahoo messenger example, use anchors and place identifying prefix string in the first position after the anchor

("^(*ymsg*|*ypns*|*yhoo*)" for the Yahoo pattern). For an input that does not start with a valid prefix of the pattern, we can easily tell from the beginning of the input that it will not match the pattern. Patterns without an anchor, however, force us to check every byte in the input to make such a conclusion.

Hard-to-match patterns

Generally speaking, patterns with many wildcards ("." and "*") are hard to match. This is because wildcards will interact with other wildcards to form large DFA components, when using DFA-based approaches. For NFA-based approaches, wildcards will activate multiple states simultaneously, causing the processing complexity to increase dramatically. Similarly, patterns with classes of characters "[]" are also be hard to match because it yet is another form of wildcards.

The most complicated patterns we encountered in the real world are those with semantic ambiguity. If there are multiple ways of matching an input against a pattern, DFA-based approaches require many states to remember all the possible ways that patterns could match. This causes a high memory requirement. At the same time, NFA-base approaches will generate many active states, hence resulting in high processing complexity. A more detailed analysis of these hard-to-match patterns will be shown later in Section 6.4.1.

6.3 Regular Expression Matching in Network Scanning applications

The previous section surveyed the most representative traditional approaches to regular expression matching. In this session, we show that these techniques sometimes do not work efficiently for the patterns in networking applications. We begin this session with the special matching requirements of networking applications. Next, we study the typical structures of patterns in the networking applications. After that, we show that certain structures in

networking applications patterns are hard to match using traditional methods: they either require a large memory usage or yield a high computation cost. Finally we present our problem statement.

6.3.1 Requirements and Design Considerations

Unlike traditional pattern matching applications, e.g., searching for a pattern in a local file, network applications have unique requirements for pattern matching. First, the target is a continuous input stream rather than a fixed length string. Second, matching must be performed at high speed. Third, the memory consumption must be within the capacity and budget of modern computers or routers. Finally, the pattern matching scheme must be resilient to malicious attacks. In this section, we explain these points in detail.

6.3.1.1 Identify Patterns in an Input Stream

Most existing studies of regular expressions focus on a specific type of evaluation, that is, checking if a fixed length string belongs to the language defined by a regular expression. More specifically, a fixed length string is said to be in the language of a regular expression, if the string is matched from *start* to *end* by a DFA corresponding to that regular expression. In contrast, in packet payload scanning, a regular expression pattern can be matched by the entire input or specific *substrings* of the input. Without a priori knowledge of the starting and ending positions of those substrings (unless the pattern starts with "^" that restricting it to be matched at the beginning of the input), the DFAs created for recognizing all substring matches can be highly complex. This is because the DFA needs to remember all the possible sub-prefixes it has encountered. When there are many patterns with a lot of wildcards, they can be simultaneously active (recognizing part of the pattern). Hence, a DFA needs many states to record all possible combinations of partially matched patterns.

For a better understanding of the matching model, we next present a few concepts pertaining to the completeness of matching results and the DFA execution model for substring matching. Given a regular expression pattern and an input string, a complete set of results contains all substrings of the input that the pattern can possibly match. For example, given a pattern *ab** and an input *abbb*, three possible matches can be reported, *ab, abb,* and *abbb*. We call this style of matching Exhaustive Matching. It is formally defined as below:

Exhaustive Matching: Consider the matching process M as a function from a pattern P and a string S to a power set of S, such that, $M(P, S)$ = {substring S' of S| S' is accepted by the DFA of P}.

In practice, it is expensive and often unnecessary to report all matching substrings, as most applications can be satisfied by a subset of those matches. For example, if we are searching for the pattern for the Oracle user name buffer overflow attempt *"^USR\s[^\n]{100,}",* which searches for packets starting with "*USR\s*" and followed by 100 or more non-return characters. An incoming packet with "*USR\s*" followed by 200 none return characters, may have 100 ways of matching the pattern because each combination of the "*USR\s"* with the sequential 100 to 200 characters is a valid match of the pattern. In practice, reporting just one of the matching results is sufficient to detect the buffer overflow attack. Therefore, we propose a new concept, *Non-overlapping Matching,* that relaxes the requirements of exhaustive matching.

Non-overlapping Matching: Consider the matching process M as a function from a pattern P and a string S to a set of strings, specifically, $M(P, S)$ = {substring S_i of S| $\forall S_i, S_j$ accepted by the DFA of P, $S_i \cap S_j = \phi$ }.

If a pattern appears in multiple locations of the input, this matching process reports all non-overlapping substrings that match the pattern. We revisit our example above. For the pattern *ab** and the input *abbb*, the three matches overlap by sharing the prefix *ab*. For this

example, if we assume non-overlapping matching, we only need to report one match instead of three.

For most payload scanning applications, we expect that non-overlapping matching would suffice, as those applications are mostly interested in knowing if certain attacks or application layer patterns appear in a packet. In fact, most existing scanning tools like grep and Flex and systems like SNORT [10] and Bro [28] implement special cases of non-overlapping matching such as left-most longest matching or left-most shortest matching. As we show later this section, by restricting the solutions for non-overlapping matching, we can construct more memory-efficient DFAs.

Note that it is possible for patterns to be fragmented into multiple packets since each packet has a limited size. In SNORT and Bro, packets are passed to a *defragmenter* before entering the pattern matching system. We also assume having such a defragmentation component as explained earlier in Section 1.4.2.

6.3.1.2 High Speed

The above section discussed that we need to matching strings inside data stream. The rate of the input packet stream is usually at at least 1 Gigabit rate and hence the packet payload matching must also be performed at least at 1 Gigabit rate. In this section, we map this requirement into the regular expression matching process and explore the tradeoff of different matching methods.

As discussed in Section 6.2.2, NFA-based approaches have $O(n^2)$ processing complexity given an input character for a regular expression of length n, while DFA-based approaches have a significantly lower computation cost of $O(1)$. Hence, in this chapter, we adopt DFA-based approaches to achieve a high matching speed. There are two DFA-based automata

processing: repeated searches and one pass search. Next we explain these processing approaches in detail.

Repeated searches. A DFA can be created directly from a pattern using standard DFA construction techniques [100, 101]. To find the set of matching substrings (using either exhaustive or non-overlapping matching), the DFA execution needs to be augmented with repeated searches of the input: an initial search starts from the beginning of the input, reading characters until (1) it has reported all matches (if exhaustive matching is used) or one match (if non-overlapping matching is used), or (2) it has reached the end of the input. In the former case, the new search will start from the next character in input (if exhaustive matching is used) or from the character after the reported match (if non-overlapping matching is used). In the latter case, a new search is initiated from the next character in the input. This style of repeated scanning using DFAs is commonly used in language parsers. However, this repeated scanning process is slow for packet payload scanning where the probability of the packet payload matching any particular pattern is low (verified in Section 6.6.6).

One-pass search. In the second approach, ".*" is pre-pended to each pattern without '^', which explicitly states that the pattern can be matched anywhere in the input. Then, a DFA is created for the extended pattern. As the input is scanned from start to end, the DFA can recognize all substring matches that may start at different positions of the input. Using one pass search, this approach can truly achieve $O(1)$ computation cost per character, thus suitable for networking applications. To achieve a high scanning rate, we adopt this approach in the rest of the study. This approach, however, may generate a larger number of DFA states compared one based on repeated searches. We will address this problem in Section 6.4 and 6.5, with Section 6.4 emphasizing on matching a single regular expression and Section 6.5 on multiple ones. Later, we will compare the performance this one-pass search approach and the repeated search approach in Section 6.6.6

6.3.1.3 Low Memory Requirements

Packet processing systems usually have limited memory, especially limited fast memory. The most popular fast memory used in routers today is Static Random Access Memory (SRAM). The largest single-chip SRAM currently available from Cypress is only 72Mb [91]. The size of slow memory, i.e., Dynamic random access memory (DRAM), can easily go to Giga-bytes. However, its access latency is ten times that of SRAM, which makes it a distant second choice for fast regular expression matching purpose. Even if a large DRAM is adopted, it still has some fixed capacity. The exponential growth of DFA can easily eat this up. Thus, the focus of our approach is to reduce memory overhead of DFA while approaching the optimal processing speed of $O(1)$ per character.

It is worth noting that there are two sources of memory usage in DFAs: states and transitions. The number of transitions is linear with respect to the number of states because for each state there can be at most 2^8 (for all ASCII characters) links to next states. Therefore, the number of states (in the minimized DFA) is the primary factor in determining the memory usage in the rest of the chapter. Also, due to the need for high performance, DFAs using any table compression techniques is not adopted because such techniques incur extra memory accesses per input character and thus slow the matching process.

6.3.1.4 Resilience to Attacks

Many network packet scanning applications are designed for intrusion detection purposes, i.e., for filtering out virus or worm attack packets. Therefore, the regular expression scanning system itself must be resilient to attacks. Most existing packet scanning systems are optimized for common traffic. They spend a longer time or use more memory on packets that contain complicated patterns or infrequent appearing patterns. There are two types of attacks to these systems. The first is overloading attacks. Intruders (e.g., SNORT system [91]) attack

these systems by maliciously sending a high volume of packets that require heavy processing. When the intrusion system slows down and starts to drop the incoming packets, a malicious packet can sneak through. The second type of attack is to crash the system. Intruders maliciously send packets that require a lot of dynamic memory, thus causing the system to crash.

To be resilient to attacks, the pattern matching system must guarantee good throughput even under worst case traffic. In other words, it must deliver deterministic performance both in terms of processing complexity and memory usage.

6.3.2 Patterns Used in Networking Applications

Having identified the design goals, next we take a detailed look at the patterns in networking applications. We study the complexity of DFA for typical patterns used in real-world packet payload scanning applications such as Linux L7-filter (as of Feb 2006), SNORT (Version 2.4), and Bro (Version 0.8V88). The study is based on the use of *exhaustive matching* and *one-pass search* as we presented in Subsections 6.3.1.1 and 6.3.1.2. Table 6-3 summarizes the results.

Explicit strings generate DFAs of size linear to the number of characters in the pattern. 25% of the networking patterns, in the three applications we studied (Linux L7-filter, SNORT, and Bro), fall into this category and they generate relatively small DFAs with an average of 24 states. Usually patterns starting with the anchor "^" generate small DFAs, while patterns with wildcards generate large ones. However, patterns in networking applications may not follow this form. For example, a buffer overflow attempt string may be hidden in a series of valid commands. Wildcards ".*" alone need not create a large number of states. 19% of the patterns contain wildcards, but still generate a small number of states (on average 27). In these patterns, usually there is only a single wildcard and it doesn't

Table 6-3. An analysis of patterns in network scanning applications.

Pattern features	Example	# of states	% of patterns	Average # of states
1) Explicit strings with *k* characters	*^ABCD* *.*ABCD*	*k+1*	25.1%	*23.6*
2) Wildcards	*^AB.*CD* *.*AB.*CD*	*k+1*	18.8%	*27.2*
3) Patterns with ^, a wildcard, and a length restriction *j*	*^AB.{j+}CD* *^AB.{0,j}CD* *^AB.{j}CD*	*O(k*j)*	44.7%	*180.3*
4) Patterns with ^, a class of characters overlaps with the prefix, and a length restriction *j*	*^A+[A-Z]{j}*D	*O(k+j2)* *j~370*	5.1%	*136903*
5) Patterns with a length restriction *j*, where a wildcard or a class of characters overlaps with the prefix	*.*AB.{j}CD* *.*A[A-Z]{j+}*D	$O(k+2^j)$ *j~344*	6.3%	*>2214*

interact with other parts of the pattern. For patterns starting with "^', they create DFAs of polynomial complexity with respect to the pattern length *k* and the length restriction *j*. Our observation from the existing payload scanning rule sets is that the pattern length *k* is usually limited. The length restriction *j* is usually small too, unless it is for buffer overflow attempts. In that case, it will be more than 300 hundred on average and sometimes even reaches thousands. Given large *j*, some patterns starting with anchors (Case 4) can also result in a large DFA with the number of state grows quadratic in *j*. Although this type of patterns only constitutes 5.1% of the total patterns, they create DFAs with an average of 136903 states. There are also a small percent (6.8%) of patterns starting with ".*" and having length restrictions (Case 5). These create DFAs of exponential sizes. We will address these two cases in detail later in Section 6.4.

We compare the regular expressions used in these three networking applications, SNORT, Bro, and the Linux L7-filter, against those used in emerging Extensible Markup Language (XML) filtering applications [101, 102] where regular expressions are matched over text documents encoded in XML, see Table 6-4. We notice three main differences:

Table 6-4. Comparison of regular expressions in networking applications against those in XML filtering.

	SNORT	Bro	L7-filter	XML filtering
# of regular expressions analyzed	1555	2780	70	1,000-100,000
% of patterns starting with "^"	74.4%	2.6%	72.8%	≥80%
% of patterns with wildcards ".*, +, ?, *"	74.9%	98.8%	75.7%	50% - 100%
Average # of wildcards per pattern	4.7	4.3	7.0	1-2
% of patterns with class "[ ]"	31.6%	65.8%	52.8%	0
Average # of classes per pattern	8.0	3.4	4.8	0
% of patterns with length restrictions on classes or wildcards	56.3%	23.8. %	21.4%	≈0

(1) While both types of applications use wildcards ('.', '?', '+', '*'), the patterns for packet scanning applications contain larger numbers of them. Many such patterns use multiple wildcard *metacharacters* (e.g., '.', '*'). For example, the pattern for identifying the Internet radio protocol, "*membername.*session.*player*", has two wildcard fragments ".*". Some even contain over ten such wildcard fragments. For example, the pattern for a Domain Name Server (DNS) request is "*^.?.?.?.?[\x01\x02].?.?.?.?.?.?[\x01-?][a-z0-9][\x01-?a-z]*[\x02-\x06][a-z][a-z][fglmoprstuvz]?[aeop]?(um)?[\x01\x10\x1c] [\x01\x03\x04\xFF]*", which contains 14 wildcard fragments. As regular expressions are converted into state machines for pattern matching, large numbers of wildcards can cause the corresponding DFAs to grow exponentially.

(2) Classes of characters ("[]") are used in packet scanning applications, but not in XML processing applications. For example, the pattern for matching the ftp protocol, "*^220*[*\x09-\x0d -~*]**ftp*", contains a class (inside the brackets) that includes all the printing characters and space characters. The class of characters may intersect with other classes or wildcards. For example, the pattern for detecting buffer overflow attacks to the Network News Transport Protocol (NNTP) is "*^SEARCH\s+*[*^\n*]*{1024}*", where a class of character "[*^\n*]"

interacts with its preceding white space characters "\s+". When given an input with SEARCH followed by a series of white spaces, there is ambiguity whether these white spaces match "\s+" or the none return class "[^\n]". As we will show later in the Case 4 of Section 6.4.1, such interaction can result in a highly complex state machine.

(3) A high percentage of patterns in packet payload scanning applications have length restrictions on some of the classes or wildcards, while such length restrictions usually do not occur in XML filtering. For example, the pattern for detecting Internet Message Access Protocol (IMAP) email server buffer overflow attack is as follows ".*$AUTH\s[^\n]\{100\}$". This pattern contains the restriction that there would be 100 non-return characters "[^\n]" after matching of keyword *AUTH* and any number of white spaces "\s". As we shall show later in the Case 5 of Section 6.4.1, such length restrictions can increase the resource needs for expression matching.

The above study demonstrates that, compared to the XML filtering application, network packet scanning applications face additional challenges. These challenges lead to a significant increase in the complexity of regular expression matching, as we will show later in detail in Section 6.4.1.

6.3.3 Problem Statement

The last subsection shows that patterns in networking applications have uniquely complex features that can lead to complex processing using traditional methods. In this chapter, we seek a fast and memory-efficient solution to regular expression matching for packet payload scanning. The scope of the problem is defined as follows:

We consider ***DFA-based approaches***, as NFA-based approaches are computationally inefficient on serial processors or processors with limited parallelism (e.g., multi-core CPUs in comparison to FPGAs). Our goal is to achieve $O(1)$ computation cost for each incoming

character, which cannot be accomplished by any existing DFA-based approaches due to their excessive memory usage. Thus, the focus of the study is to give an approach that reduces memory overhead from that of a traditional DFA-based approach while approaching the optimal processing speed of $O(1)$ per character.

This chapter focuses on algorithm designs geared towards ***network processor and general-purpose processor-based architectures,*** and explores the limits of regular expression matching in these environments. Wherever appropriate, it leverages of the parallel processing capabilities of multi-core processors, which are rapidly becoming prevalent in those architectures. Nevertheless, the results can be used in FPGA-based and ASIC-based approaches as well [103]. Note that the number of cores and the amount of local memory in the multi-core processors are usually limited. For example, the IBM cell processor has 8 cores, each with only 128 KB local memory [53] .

As stated in Section 6.3.1, our goal is to design a fast regular expression matching system that has $O(1)$ per character with minimum memory requirement.

6.4 Matching of Individual Patterns

In this section, we present our algorithm for matching individual regular expression patterns. The main technical challenge is to create DFAs that can fit in memory, thus making a fast DFA-based approach feasible. We first analyze the size of DFAs for hard-to-match patterns in typical payload scanning applications in Section 6.4.1. Although theoretical analyses [100, 101] have shown that DFAs are subject to an exponential increase in state as the pattern size increases, here, we identify *specific* structures that can lead to exponential growth of DFAs. Based on the insights from this analysis, in Section 6.4.2, we propose pattern rewrite techniques that explore the possibility of trading off exhaustive pattern matching (defined in

Section 6.3.1.1) for memory efficiency. Finally, we offer guidelines to pattern writers on how to write patterns amenable to efficient implementation in Section 6.4.3.

6.4.1 DFA Analysis for Individual Regular Expressions

In Section 6.3.2, we analyzed the typical patterns in networking applications. A large percentage of them are easy to match. There are two exceptions, one case generates DFAs of quadratic size (Case 4 of Table 6-3) and the other generates exponential-size DFAs (Case 5 of Table 6-3). Next, we explain these two cases in more detail.

DFAs of Quadratic Size

A common misconception is that patterns starting with '^' create simple DFAs. However, even with '^', classes of characters that overlap with the prefix pattern can still yield a complex DFA. Consider the pattern ^*B*+[^*n*]*{3}D*, where the class of character [^*n*] denotes any character but the return character (*n*). Figure 6-1 shows that the corresponding DFA has a quadratic number of states. The quadratic complexity comes from the fact that the letter *B* overlaps with the class of character [^*n*] and, hence, there is inherent ambiguity in the pattern: A second *B* letter can be matched either as part of *B+,* or as part of *[^\n]{3}*. Therefore, if an input contains multiple *B*s, the DFA needs to remember the number it has seen and their locations i to make a correct decision with the next input character. If the class of characters has length restriction of *j* bytes, DFA needs $O(j^2)$ states to remember the combination of distance to the first *B* and the distance to the last *B*.

Seventeen patterns in the SNORT rule set fall into this quadratic states category. For example, the regular expression for the NNTP rule is "^*SEARCH**s*+[^*n*]{*1024*}". Similar to the example in Figure 6-1, *s* overlaps with ^*n*. White space characters cause ambiguity of whether they should match *s*+ or be counted as part of the 1024 non-return characters [^*n*]{*1024*}. Specifically, an input of *SEARCH* followed by 1024 white spaces and then 1024

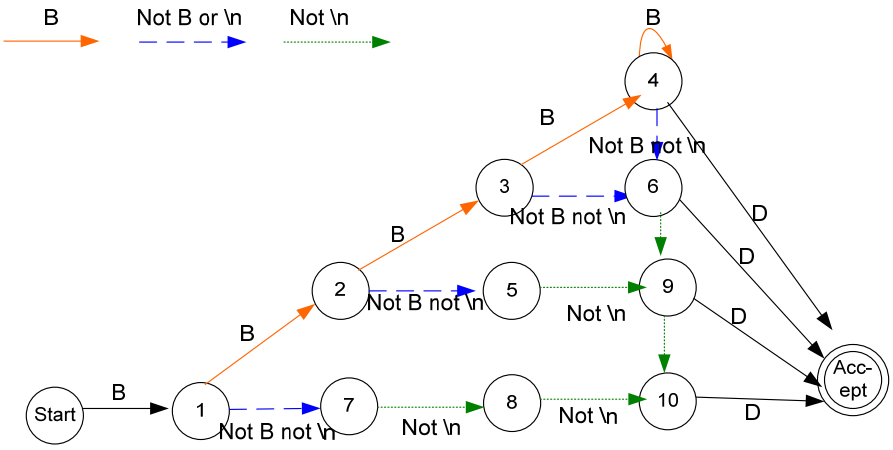


Figure 6-1. A DFA for pattern ^*B*+[^*n*]{*3*}*D* .

'a's will have 1024 ways of matching strings, i.e., one white space matches \s+ and the rest as part of [^\n]{*1024*}, or two white spaces match \s+ and the rest as part of [^\n]{*1024*}, and so on. By using 1024^2 states to remember all possible sequences of these white spaces, the DFA accommodates all the ways to match the substrings of different lengths. Note that all these substrings start with *SEARCH* and hence are overlapping matches.

This type of quadratic state problem cannot be solved by an NFA-based approach. Specifically, the corresponding NFA contains 1042 states; among these, one is for the matching of *SEARCH*, one for the matching of \s+, and the rest of the 1024 states for the counting of [\^n]{*1024*} with one state for each count. An intruder can easily construct an input as "*SEARCH*" followed by 1024 white spaces. With this input, both the \s+ state and all the 1023 non-return states would be active at the same time. Given the next character, the NFA needs to check these 1024 states sequentially to compute a new set of active states.

This problem cannot be solved by a fixed string pre-filtering scheme (as used by SNORT), which first identifies the fixed string and then directs the suspicious packets to a second level checking. Pre-filtering can only recognize the presence of the fixed string "*SEARCH*" in the input. After that, an NFA or DFA-based matching scheme is still needed in post processing to check whether the input matches the pattern. Another choice is to count

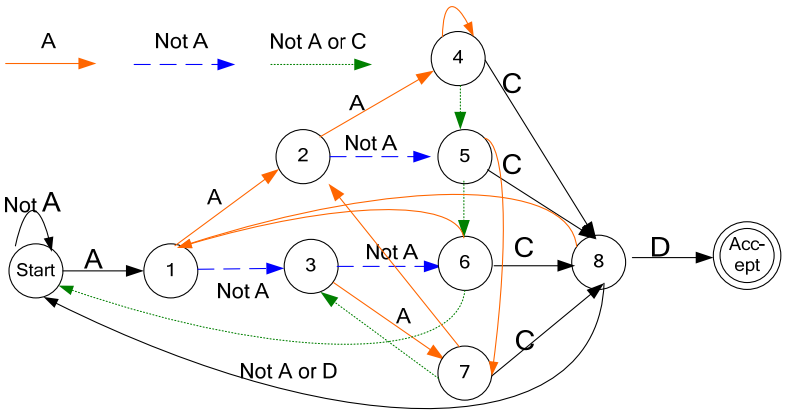


Figure 6-2. A DFA for pattern *.*A.{2}CD .*

the subsequent characters in post processing after identifying the prefix "*SEARCH*". This approach does not solve the problem because every packet (even normal traffic) with the prefix will incur the counting process. In addition, intruders can easily construct packets with multiple (different) prefixes to invoke many requests for such post processing.

DFA of Exponential Size

The previous case presents the patterns leading to quadratic DFA sizes. Next, we study patterns generating exponential DFA sizes. In real life, many payload scanning patterns contain an ***exact distance*** requirement. Figure 6-2 shows the DFA for an example pattern ".**A..CD*". An exponential number of states (2^{2+1}) are needed to represent these two wildcard characters. This is because we need to remember all possible effects of the preceding *A*s as they may yield different results when combined with subsequent inputs. For example, an input *AAB* is different from *ABA* because a subsequent input *BCD* forms a valid pattern with *AAB* (*AABBCD*), but not so with *ABA* (*ABABCD*). In general, if a pattern matches exactly *j* arbitrary characters, $O(2^j)$ states are needed to handle the requirement that the distance *exactly equals j* . This result is also reported in [101]. Similar results apply to the case where the class of characters overlaps with the prefix, e.g., ".**A*[*A-Z*]{*j*}D".

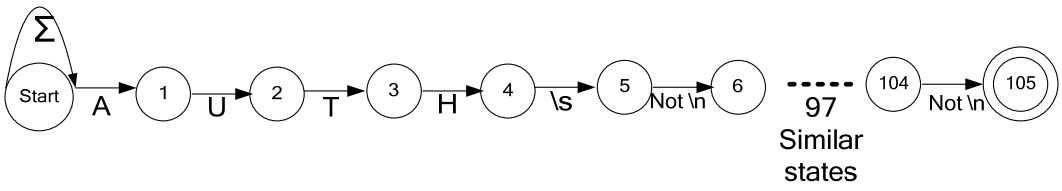


Figure 6-3. NFA for the pattern .AUTH\s*[^*n*]{*100*}** .

Similar structures exit in real world pattern sets. In the intrusion detection system SNORT, 53.8% of the patterns (mostly for detecting buffer overflow attempts) contain a fixed length restriction. Around 80% of the rules start with ^; hence, they will not cause exponential growth of DFA. The remaining 20% of the patterns do suffer from the state explosion problem. For example, consider the rule for detecting IMAP authentication overflow attempts, which uses the regular expression ".*AUTH\s[^\n]{100}". This rule detects any input that contains *AUTH*, then a white space, and no return character in the following 100 bytes. If we directly compile this rule into a DFA, the DFA will contain more than 10,000 states because it needs to remember all the possible consequences that an *AUTH\s* subsequent to the first *AUTH\s* can lead to. For example, the second *AUTH\s* can either match [^\n]{100} or be counted as a new match of the prefix of the regular expression.

It is obvious that the exponential blow-up cannot be mitigated by using an NFA-based approach. Figure 6-3 shows the NFA for the pattern ".*AUTH\s[^\n]{100}". Because the first state has a self-loop marked with Σ, the input "*AUTH\sAUTH\sAUTH\s*..." can cause a large number of states to be simultaneously active, resulting in significantly degraded system performance, as demonstrated by our results reported in Section 6.6.6. Another approach is clearly needed to achieve fast regular expression matching without causing memory explosion.

6.4.2 Regular Expression Rewrites

The previous section has identified the typical patterns that yield large DFAs. This section investigates the possibility of rewriting some of those patterns to reduce the DFA size. Such rewriting is enabled by relaxing the requirement of exhaustive matching to that of *non-overlapping matching*. In particular, we propose two rewrite rules, one for rewriting specific patterns belonging to the case of quadratic-sized DFAs (Case 4 in Section 6.4.1), and the other for rewriting specific patterns that generate exponential-sized DFAs (Case 5 of Section 6.4.1). Those patterns amenable to rewrites have the following characteristic. Their have a class of characters with length restrictions that overlap with their prefixes. Taken the previous example ".**AUTH*\s[^\n]{*100*}", *"[^\n]"* has a length restriction and it overlaps with "*AUTH"* . These patterns are typical in real-world rulesets such as SNORT and Bro. For these patterns, as shown in Section 6.4.1, neither the NFA-based approaches nor the fixed string pre-filtering scheme can handle them efficiently. In contrast, the rewrites rules can convert these patterns into DFAs with their sizes successfully reduced from quadratic or exponential to only linear.

Rewrite Rule (1)

As shown in Section 6.4.1, patterns that start with '^' and contain classes of characters with length restrictions, e.g., "^*SEARCH**s*+[^*n*]{*1024*}", can generate DFAs of quadratic size with respect to the length restriction. Below, we first explain the intuition behind Rewrite Rule (1) using the above example and then state and prove a theorem for more general cases.

We explained in Section 6.3.3.1 that we want to identify non-overlapping patterns. Under this assumption, pattern "^*SEARCH**s*+[^*n*]{*1024*}" can be rewritten to "^*SEARCH**s*[^\n]{*1024*}". The new pattern specifies that after matching *SEARCH* and a single white space *s*, it starts counting non-return characters for [^*n*]{*1024*} regardless of the content. In this way, the ambiguity of matching *s* is removed. Hence, this pattern requires

a number of states linear in the length restriction *j* (1024 in the example). Although this new pattern greatly reduces the number of states, it is still equivalent to the original pattern in identifying non-overlapping patterns. This is because the new pattern essentially implements non-overlapping left-most shortest match. Next, we provide a theorem to prove the equivalence of two patterns.

Theorem 1: Pattern *"^A[A-Z]{j}"* is equivalent to the original pattern *"^A+[A-Z]{j}"* for detecting non-overlapping shortest string.

Theorem 1 (proved in Appendix A) is for a more general case where the suffix of a pattern contains a class of characters overlapping with its prefix and a length restriction, "^*A*+[*A-Z*]{*j*}". The theorem proves that this type of pattern can be rewritten to "^*A*[*A-Z*]{*j*}" with equivalence guaranteed under the condition of non-overlap matching. Note that our rewrite rule can also be extended to patterns with various types of length restriction such as "^*A*+[*A-Z*]{*j*+}" and "^*A*+[*A-Z*]{*j*,*k*}".

Using Rewrite Rule (1), 17 similar patterns in the SNORT rule set can be rewritten to patterns with smaller DFAs. Detailed results regarding these rewrites are reported in Section 6.6.2.

Rewrite Rule (2)

As discussed in Section 6.4.1, patterns like ".**AUTH**s*[^*n*]{*100*}" generate exponential numbers of states to keep track of all the *AUTH\s* subsequent to the first *AUTH\s*. If non-overlapping matching is used, the intuition of our rewriting is that after matching the first *AUTH\s*, we do not need to keep track of the second *AUTH\s*. This is because:

- If there is a '*n*' character within the next 100 bytes, the return character must also be within 100 bytes to the second *AUTH\s*, and
- If there is no '*n*' character within the next 100 bytes, the first *AUTH\s* and the following characters have already matched the pattern.

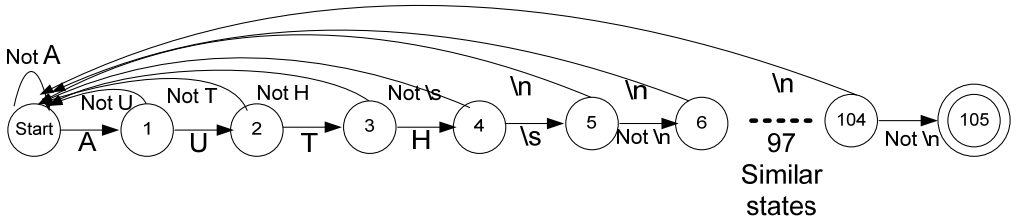


Figure 6-4. DFA for rewriting the pattern .AUTH\s*[^*n*]{*100*} .**

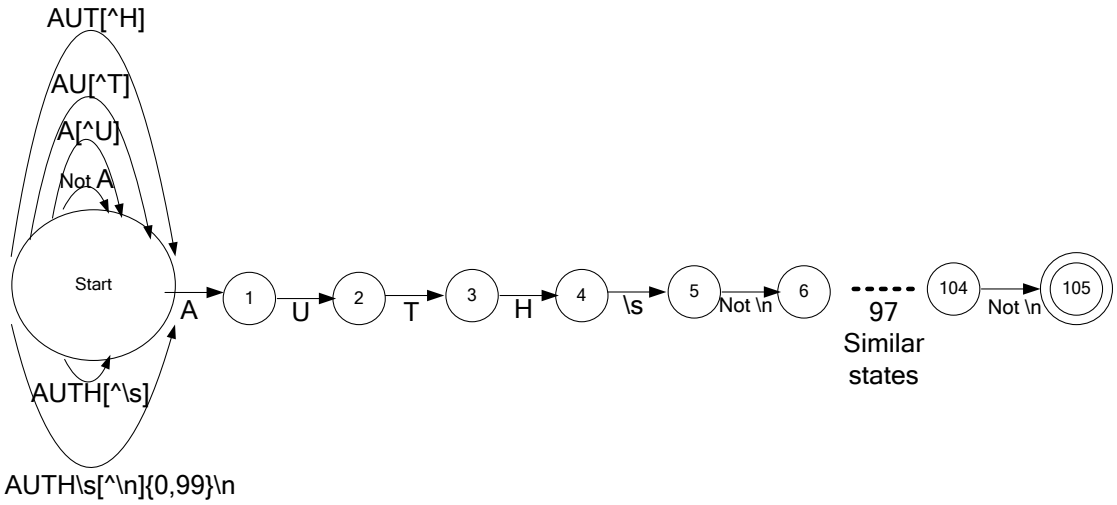


Figure 6-5. Transformed NFA for deriving Rewrite Rule (1) .

The intuition is that we can rewrite the pattern such that it only attempts to capture one match of the prefix pattern. Following the intuition, we can simplify the DFA by removing the states that deal with the successive *AUTH\s*. As shown in Figure 6-4, the simplified DFA first searches for *AUTH* in the first 4 states, then looks for a white space, and after that starts to count and check whether the next 100 bytes contains a return character. After rewriting, the DFA only contains 106 states.

The rewrite pattern can be derived from the simplified DFA shown in Figure 6-4. Applying a standard technique that maps a DFA/NFA to a regular expression [100], we can transform this DFA to an equivalent NFA in Figure 6-5. For the link that moves from state 1

back to the start state in Figure 6-4 (i.e., matching *A* then not *U*), the transformed NFA places it right at the start state and labels it with *A*[^*U*]. The transformed NFA does the same for each link moving from state *i* ($1 \le i \le 105$) to the start state in Figure 6-4. The transformed NFA can be directly described using the following regular expression:

"([^*A*]|*A*[^*U*]|*AU*[^*T*]|*AUT*[^*H*]|*AUTH*[^*s*]|*AUTH**s* [^*n*]{*0,99*}*n*)**AUTH**s*[^*n*]{*100*}".

This rule first enumerates all the cases that do not satisfy the pattern and then attaches the original pattern to the end of the new pattern. In other words, ".*" is replaced with the cases that do not match the pattern, represented by:

([^*A*]|*A*[^*U*]|*AU*[^*T*]|*AUT*[^*H*]|*AUTH*[^*s*]|*AUTH**s*[^*n*]{*0,99*}*n*)*.

Then, when the DFA comes to the states for *AUTH**s*[^*n*]{*100*}, it must be able to match the pattern. Since the rewritten pattern is directly obtained from a DFA of size *j+5*, it generates a DFA of a linear number of states rather than an exponential number before applying the rewrite.

Theorem 2. Pattern ".**AB[A-Z]{j}*" can be rewritten as "*([^A]|A[^B]|AB[A-Z]{j-1}[^(A-Z)])*AB[A-Z]{j}*". These two patterns are equivalent for detecting non-overlapping strings.

For a more general case ".**AB*[*A-Z*]{*j*}", Theorem 2 (proved in Appendix B) states the equivalence of the new pattern and the original pattern under the condition of non-overlapping matching. Moreover, it offers rewrite rules for patterns in other forms of length restriction, e.g., ".**AB*[*A-Z*]{*j+*}".

Rewrite Rule (2) is applicable to 54 expressions in the SNORT rule sets and 49 in the Bro rule set. We wrote a script to automatically rewrite these patterns and observed significant reduction in DFA size. Detailed simulation results are reported in Section 6.5.2.

6.4.3 Guidelines for Pattern Writers

As mentioned above, an important implication of this work is that the pattern rewriter can automatically perform both types of rewriting. An additional benefit is that our analysis provides insight into how to write regular expression patterns that are amenable to efficient DFA implementation.

From the analysis in Section 6.4.1, we can see that sometimes patterns with length restrictions can generate large DFAs. There are two categories. One is starting with an anchor, but length restriction overlaps with the proceeding class. The other type is patterns without anchor but containing length restrictions.

In typical packet payload scanning pattern sets including Linux L7-filter, SNORT, and Bro, 21.4-56.3% of the length restrictions are associated with classes of characters. The most common of these are "[^*n*]", "[^\]]" (not ']'), and "[^\]"" (not "), used for detecting buffer overflow attempts. The length restrictions of these patterns are typically large (233 on the average and reaching up to 1024). For these types of patterns, we highly encourage the pattern writer to add "^" so as to avoid the exponential state growth as shown in Section 6.4.2. For patterns that cannot start with "^", the pattern writers can use the Rewrite Rule 2 to generate patterns with linear numbers of states to the length restriction requirements.

Even for patterns starting with "^", we need to avoid the interactions between a character class and its preceding character, as shown in Rewrite Rule 1. One may wonder why a pattern writer uses *s*+ in the pattern "^*SEARCH**s*+[^\n]{*1024*}", when it can be simplified as *s*. Our understanding is that, in reality, a server implementation of a search task usually interprets the input in one of the two ways: either skips a white space after *SEARCH* and takes the following up to 1024 characters to conduct a search, or skips all white spaces and takes the rest for the search. The original pattern writer may want to catch intrusion into systems of either implementation. However, the way the original pattern is written generates

false positives if the server does the first type of implementation (skipping all the white spaces). This is because if an input is followed by 1024 white spaces and then some non-whitespace regular command of less than 1024 bytes, the server can skip these white spaces and take the follow-up command successfully. However, this legitimate input will be caught by the original pattern as an intrusion because these white spaces themselves can trigger the alarm. To catch attacks to this type of server implementation, while not generating false positives, we need the following pattern.

"^*SEARCH*\s+[^\s][^\n]{*1023*}"

In this pattern, \s+ matches all white spaces and [^\s] means the first non-white space character. If there are more than 1023 non-return characters following the first non white space character, it is a buffer overflow attack. By adding [^\s], the ambiguity in the original pattern is removed; given an input, there is the only way of matching each packet. As a result, this new pattern generates a DFA of linear size.

To generalize, we recommend pattern writers to avoid all the possible overlaps between the neighboring segments in the pattern. Here, overlap denotes an input can match both segments simultaneously, e.g., \s+ and [^\n]. Overlaps will generate a large number of states in a DFA because the DFA needs to enumerate all the possible ways to match the pattern.

6.5 Selective Grouping of Multiple Patterns

The previous section focused on how certain patterns can lead to exponential DFA sizes and presented two rewrite techniques that limit the DFA growth for such patterns to be sub-exponential. As mentioned in Section 6.2.2, it is well known that the computation complexity for processing m patterns reduces from $O(m)$ to $O(1)$ per character, when the m patterns are compiled into a single composite DFA. However, it is usually infeasible to compile together a large set of patterns due to the complicated interactions.

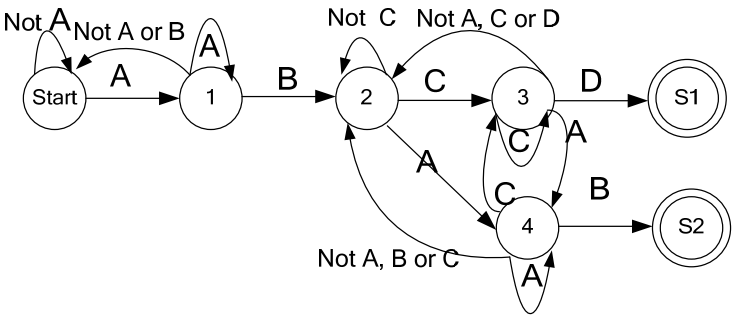


Figure 6-6. A DFA for pattern *.*ABCD* and *.*ABAB* .

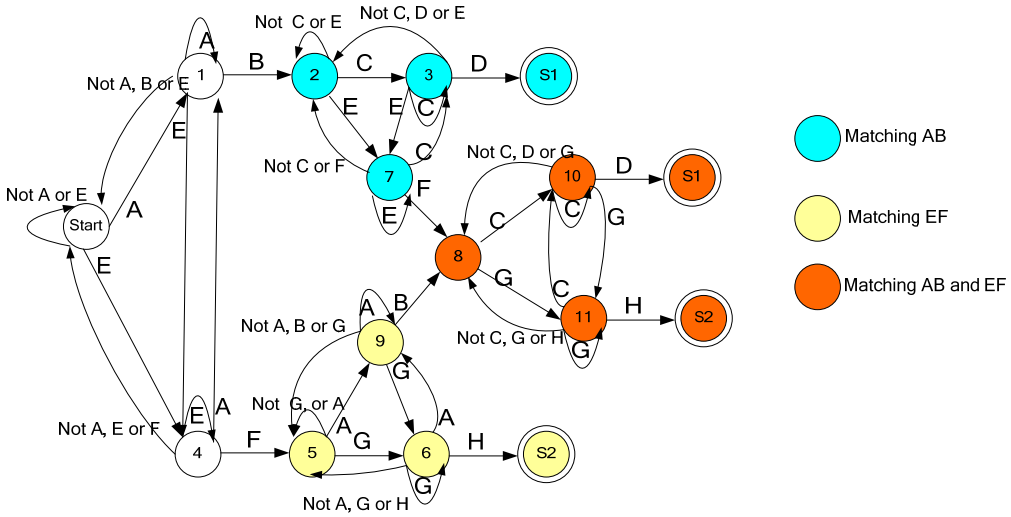


Figure 6-7. A DFA for pattern *.*AB.*CD* and *.*EF.*GH* .

We first present two examples illustrating the interactions between patterns in Subsection 6.5.1. Then we use a real-world payload scanning ruleset to demonstrate that exponential growth happens for real-world rule sets in Subsection 6.5.2. Based on these observations, we propose a new class of grouping algorithms that selectively divide patterns into groups while avoiding the adverse interaction among patterns in Subsection 6.5.3.

6.5.1 Interactions of Regular Expressions

When patterns share prefixes, some states can be merged. For example, states 1 and 2 shown in Figure 6-6 are shared by ".**ABCD*" and ".**ABAB*". Combining these patterns can save

both storage and computation. However, if the patterns do not share the same prefix, thus putting *m* patterns together may generate 2^m states.

Figure 6-7 shows a composite DFA for matching ".*AB.*CD" and ".*EF.*GH". This DFA contains many states that did not exist in the individual DFAs. Among them, state 8 is created to record the case of matching both prefixes *AB* and *EF*. Generally speaking, if there are *l* patterns with one wildcard per pattern, we need $O(2^l)$ states to record the matching of the power set of the prefixes. In such scenarios, adding one more pattern into the DFA doubles its size. If there are *x* wildcards per pattern, then $(x+1)^l$ states are required. 74.1% of patterns in the Linux L7-filter set contain two or more wildcards. For example, the pattern for the remote desktop protocol is ".*rdpdr.*cliprdr.*rdpsnd", it contains three wildcards. SNORT also has similar patterns and the number of ".*" in a pattern can be as high as six.

6.5.2 Interactions of Real-world Regular Expressions

We use the Linux L7-filter as an example to show that patterns do interact heavily with each other in real world application. If 70 patterns are compiled separately into 70 DFAs, each DFA has tens to hundreds of states. The total number of DFA states is 3533. When starting to group multiple patterns into a composite DFA (patterns are selected with a random order), the processing complexity decreases since it is no longer necessary to run the input separately through each DFA. However, the total number of DFA states (i.e., the sum of the composite DFAs generated by grouped patterns and those ungroup patterns) grows to over 136,786 states with just 40 patterns, as illustrated by the increasing dotted line in Figure 6-8. We could not add more patterns into the composite DFA because it exceeded the memory limit in the test machine that we used: 1.5 GB. However, only a certain subset of patterns cause significant DFA growth. For example, patterns 12, 37, and 38 as shown in Figure 6-8 all result in large DFAs. Similar to the previous example in Figure 6-7, these patterns all contain

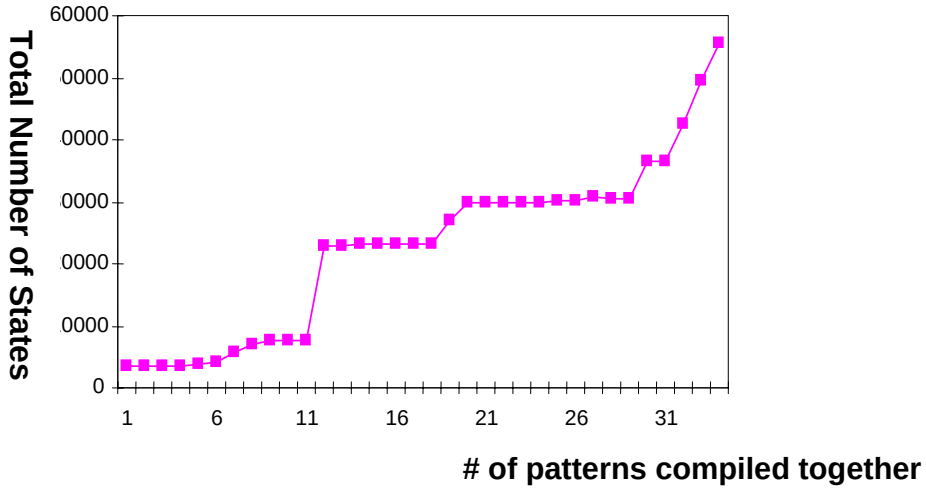


Figure 6-8. DFA size and processing complexity of multiple patterns (unsorted order, without using our grouping algorithm).

large numbers of wildcards and classes of characters "[]". We figure out that pattern 12 contains a fixed length (20) of wildcards. Pattern 37 contains three unrestricted wildcards (".*"). Pattern 38 contains 19 classes of characters, 4 unrestricted wildcards, and 8 length restricted wildcards.

6.5.3 Regular Expressions Grouping Algorithms

As discussed above, certain patterns interact with each other when compiled together, which can result in a large composite DFA. In this section, we propose algorithms to selectively partition m patterns into k groups such that the patterns in each group do not adversely interact with each other. We define *interaction* as follows: patterns interact with each other if their composite DFA contains more states than the sum of their individual ones. If m patterns can be divided into k groups, where patterns in each group do not interact with each other, we can reduce the computation complexity from $O(m)$ to $O(k)$.

To calculate the number of states in the composite DFA, an NFA is first created. Figure 6-9 shows that the NFA is constructed by adding a new start state, two ε edges leading to the

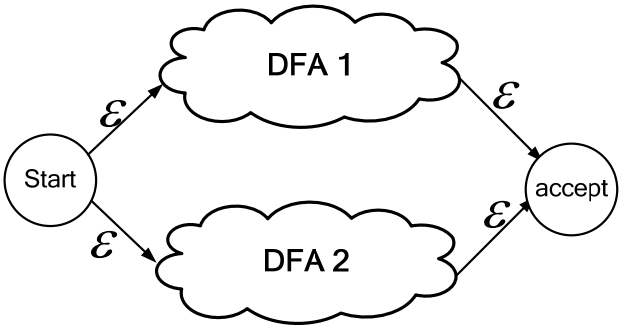


Figure 6-9. Composite NFA for two DFAs.

individual DFA, a new accepting state, and two ε edges from the DFA accepting states to the new accepting state. Then we run the NFA to DFA conversion algorithm and the DFA minimization algorithm to obtain the composite DFA.

We use the information on pairwise interaction to group a set of m regular expressions. The intuition is that if there is no interaction between any pair selected from R_1, R_2, and R_3, the composite DFA of R_1, R_2, R_3 is not likely to exceed the sum of individual ones. We validate this assumption using empirical results in Section 6.6.4.

We devise grouping algorithms both for multi-core processor architectures, where groups of patterns can be processed in parallel among different processing units, and for general processor architectures, where the DFA for one group corresponds to one process or thread. Next, we present the algorithm for the former architecture first and then the algorithm for the latter.

In hardware systems with multi-core architectures, there are multiple parallel processing units, as we introduced in Section 2.3.4. The number of cores is usually limited. Hence, one DFA per pattern per processing unit is infeasible. Our goal is to design an algorithm that divides regular expressions into several groups, so that one processing unit can run one or several composite DFAs simultaneously. In addition, the size of local memory of each

processing unit is quite limited. For example, the newly architected IBM cell processor has 8 synergistic processor elements, each with 128 KB local memory [53]. Generally speaking, the fewer the total groups, the less processing complexity, because each group requires O(1) processing for each input character. A side-effect of grouping is increasing the number of DFA states due to the interactions of patterns. Hence, our goal is to achieve the fewest groups possible under these memory limitations. In our algorithm, we continuously group patterns until they meet the local memory limit. The pseudo-code of the algorithm is provided below.

For regular expression R_i in the set
For regular expression R_j in the set
    Compute pairwise interaction of R_i and R_j
Construct a graph $G(V, E)$
V is the set of regular expressions, with one vertex per regular expression
E is the set of edges between vertices, with an edge (V_i, V_j) if R_i and R_j interact with each other.
Repeat
New group (NG) = ϕ
Pick a regular expression that has the least interaction with others and add it into new group *NG*
Repeat
Pick a regular expression *R* has the least number of edges connected to the new group
    Compile $NG \cup \{R\}$ into a DFA
    if this DFA is larger than the limit
        break;
    else
        Add *R* into *NG*
Until every regular expression in *G* is examined
Delete *NG* from *G*
Until no regular expression is left in *G*

Figure 6-10. Algorithm for Multi-core Processor Architectures with limited total memory size

In this algorithm, we first compute the pairwise interaction of regular expressions. With this pairwise information, we construct a graph with each pattern as a vertex and an edge between patterns R_i and R_j if they interact with each other. Using this graph, we can start with a pattern that has the least interaction with others, and then try to add patterns that have least interactions into the same group. We keep adding until the composite DFA is larger than the

local memory limit. Then we proceed to create a new group from the patterns that remain ungrouped.

General processor architecture. In the general processor architectures, if there are multiple composite DFAs to be run, the processor executes each of them sequentially. Usually all the DFAs are kept in the main memory for performance purposes. Since the memory is shared among all DFAs, we want to group all patterns into the smallest number of groups (hence the smallest number of DFA) while not exceeding the available memory size. Finding the smallest number of groups is an NP hard problem. In this work, we apply heuristics to find a small number of groups that can serve as a good approximation. The pseudo-code of our algorithm for the general processor architectures is shown in the following.

Leftover memory L = Total memory
For regular expression R_i in the set
Compute the DFA size D_i for R_i
Leftover memory = Leftover memory - D_i

Repeat
New group (NG) = ϕ
Pick a regular expression which has the least interaction with others and add it into new group NG
Repeat
Pick a regular expression R that has the least number of edges connected to the new group
Compile $NG \cup \{R\}$ into a DFA
if $D(NG) > \sum_{R_i \in NG} D(R_i) + L * |NG| / (\#\text{left patterns})$
break;
else
Add R into NG
Until every regular expression in G is examined
Leftover memory $L = L - D(NG) - \sum_{R_i \in NG} D(R_i)$
Delete NG from G
Until no regular expression is left in G

Figure 6-11. ***Algorithm for General-Processor Architectures***

Unlike the multi-core case, in this algorithm we first compute the DFA of individual patterns and compute the leftover memory size. At any stage, we always try to distribute the leftover memory evenly among the ungrouped expressions, which is the heuristic that we apply to increase the number of grouping operations which in turn reduces the number of resulting groups. In this algorithm, we group patterns using a similar routine as the previously. However, we stop grouping when the size of the composite DFA (denoted as $D(NG)$) exceeds its share of the leftover memory. Here, the DFA's share of the leftover memory is calculated using the formula = (Leftover memory L) * (Number of patterns in the group) / (Number of ungrouped patterns).

Discussion: Grouping multiple regular expressions into one composite DFA is a well known technique to enhance matching performance. Our algorithms focus on picking the "right" patterns to be grouped together. Here, "right" means patterns don not interact with each other exponentially; therefore grouping them together will not generate a large DFA. Similar to our approach, systems like Bro group patterns into one group, instead of partitioning them into several. They adopt a lazy DFA-based approach, where commonly used DFA states are cached and the DFA is extended at run-time if needed. The distinction between our approach and Bro's is that our grouping algorithm produces scanners of deterministic complexity (*O(1)* per input character). The lazy DFA-based approach, although fast and memory efficient on most common inputs, may be exploited by intruders to construct malicious packets that force the lazy DFA to enter many corner cases [24] (also in Section 6.3.1.4), this may result in drastically slowed performance. Our fully-developed DFA does not have performance degradation under such attacks. This is because no matter what kind of input characters a packet contains, our scheme only does one memory lookup per packet. The computation cost is deterministic to any input.

6.6 Evaluation Results

We implement a DFA scanner with rewriting and grouping functionality for fast and memory efficient regular expression matching. In this section, we evaluate the effectiveness of our rewriting techniques for reducing DFA size, and the effectiveness of the grouping algorithms for creating memory-efficient composite DFA. We also compare the speed of our scanner against a DFA-based repeated scanner generated by the widely used Flex system [104] and a best-known NFA-based scanner [60]. Compared to the DFA-based repeated scanning approach, our approach based on one pass scanning has a 12 to 42 times performance improvement. Compared to the NFA-based implementation, our DFA scanner is 50 to 700 times faster on traffic dumps obtained form MIT and Berkeley networks.

6.6.1 Experimental Setup

Certain aspects of the experimental setup in this section are similar to that given in Chapter 5. Chapter 5 focused on fixed string patterns and used the SNORT system and ClamAV rule set. This chapter focuses on regular expression patterns. We still use the SNORT rule set, but not the ClamAV rule set for the experiments, because Clam AV does not contain any regular expressions. To supplement the SNORT rule set, we add two more rule sets that contain regular expressions. The first is from the Linux layer 7 filter (as of February 2006) [22] which contains 70 regular expressions for detecting different protocols. The second one is from the Bro intrusion detection system (Version 0.8V88) [28], with a total of 2781 regular expressions.

As in Chapter 5, we use two sets of real-world packet traces are used during the experiments. The first set is the intrusion detection evaluation data set from the MIT DARPA project and the second data set is from Berkeley local LAN.

Table 6-5. Rewriting effects.

Type of Rewrite	Rule Set	Number of Patterns	Average length restriction	DFA Reduction Rate
Rewrite Rule 1: (Quadratic case)	SNORT	17	370	>98%
	Bro	0	0	0
Rewrite Rule 2: (Exponential Case)	SNORT	54	344	>99%[1]
	Bro	49	214.4	>99%[1]

We use Flex [104] to convert regular expressions into DFAs. Our implementation of the DFA scanner eliminates backtracking operations [104]. It only performs a one-pass search over the input and is able to report matching results at the position of the end of each matching substring. All the experimental results reported were obtained on PCs with 3.4 GHz CPU and 3.7 GB memory.

6.6.2 Effect of Rule Rewriting

The rewriting scheme presented in Section 6.4.1 is applied to the Linux L7-filter, SNORT and Bro pattern sets. For the Linux L7-filter pattern set, it does not identify any pattern that needs to be rewritten. For the SNORT pattern set, however, 71 rules need to be rewritten. For Bro, 49 patterns (mostly imported from SNORT) need to be rewritten using Rewrite Rule 2. For these patterns, the rewriting scheme gains significant memory savings as shown in Table 6-5. For both types of rewrite, the DFA size reduction rate is over 98%.

Seventeen patterns belong to the category for which Rewrite Rule 1 can be applied. These patterns (e.g., "^*SEARCH*\s+[^\n] {*1024*}") all contain a character (e.g., \s) that is allowed to appear multiple times before a class of characters (e.g., [^\n]) with a fixed length restriction (e.g., 1024). As discussed in Section 6.4.1, this type of pattern generates DFAs that expand quadratically in the length restriction. After rewriting, the DFA sizes decrease to

become linear in the length restriction. A total of 103 patterns need to be rewritten using Rewrite Rule 2. Before rewriting, most of them generate exponential sized DFAs that cannot even be compiled successfully. With the rewriting techniques, the collection of DFAs created for all the patterns in the SNORT system can fit into 95 MB memory, which can be satisfied in most PC-based systems.

6.6.3 Effect of Grouping Multiple Patterns

In this section, the grouping techniques are applied to regular expression sets. We will show that our grouping techniques can intelligently group patterns to boost system throughput, while avoiding extensive memory usages.

The patterns of the L7-filter can be grouped because the payload of an incoming packet is compared against all the patterns, regardless of the packet header information. For the Bro pattern set, as most rules are related to packets with specific header information, the http related patterns (a total of 648) that share the same header information is picked as the test set, as well as 222 payload scanning patterns that share the same header information. Note we do not report the results of using the SNORT rule set because its patterns overlap significantly with those of the Bro rule set.

6.6.4 Interaction of Patterns

For all three sets, a majority of patterns are non-interactive (the interactive was defined in Section 6.5.1) , particularly in Bro http patterns set where all patterns are non-interactive. As a result, most patterns in these rulesets can be combined pair-wise. This nice property offers a significant potential for the grouping algorithms to produce small numbers of groups. To achieve that, one assumption that the grouping algorithms use must be verified. As stated in

---

[1]Some patterns create too many states to be compiled. The 99% number was obtained only from observing successful rewrites.

Table 6-6. Interaction of regular expressions.

	No-interaction Pair-wise	Pair-wise No-interaction lead to No-interaction three patterns
Linux L7-filter	71.18%	99.87%
Bro http	100%	100%
Bro payload	93.3%	99.99%

Section 6.5.2, the assumption is that if three patterns are pair-wise non-interactive, it is highly likely that the size of the composite DFA will not exceed the sum of the individual sizes. We compute all the two-pattern composite DFAs and three-pattern composite DFAs and check the validity this assumption on these composites DFAs. Table 6-6 shows that this assumption is valid for over 99.8% of the cases from all three pattern sets.

6.6.5 Grouping Results

We apply the grouping algorithms to all three pattern sets to partition them into small groups. For the Bro's http pattern set, since patterns do not interact with each other, it is possible to compile all 648 patterns into one composite DFA of 6218 states. The other two sets, however, cannot be grouped into one group due to interactions (defined in Section 6.5.1). Below, the results obtained using the grouping algorithm for the multi-core architectures, where local memory is limited, are reported in Table 6-7. The results for the general processor architectures are in Table 6-8.

Table 6-7 (a) shows the results for the Linux L7-filter pattern set. We start by limiting the number of states in each composite DFA to 617, the size of the largest DFA created for a single pattern in the Linux L7-filter set. This is the base line, which determines the least amount of memory required without applying our techniques. The actual memory cost is 617 times 256 next state pointers times *log*(617) bits for each pointer, which amounts to 192 KB.

Table 6-7. Results of grouping algorithms for the multi-core architectures.

7 (a) Linux L7-filter (70 Patterns)

Composite DFA state limit	Groups	Total Number of States	Compilation Time (s)
617	10	4267	3.3
2000	5	6181	12.6
4000	4	9307	29.1
16000	3	29986	54.5

7(b) Payload patterns from Bro (222 Patterns)

Composite DFA state limit	Groups	Total Number of States	Compilation Time (s)
540	11	4868	20
1000	7	4656	118
2000	5	5430	780
6000	4	9197	1038

Considering that most modern processors have large data caches (>0.5 MB), the memory cost for a single composite DFA is comparatively small. After applying our algorithm, it generates 10 groups when the limit on the DFA size is set to 617. This shows that our grouping algorithm decreases the simultaneous active DFA from 70 (original number of patterns, without grouping) into 10 (after grouping), without additional memory requirements. When we increase the amount of available memory, there is more room for the composite DFA to record the interactions of patterns. As a result, the algorithm creates fewer (from 10 to 3) groups. As today's multi-core network processors have 8-16 engines, it is feasible to allocate each composite DFA to one processor and take advantage of parallelism.

When the total state limit is increased to 16000, the grouping algorithms can decrease the number of pattern groups from 70 (originally ungrouped) to 3. This means that, given a character, the generated packet content scanner needs to perform only three state transitions instead of the 70 state transitions that were necessary with the original ungrouped case. This results in a significant performance enhancement (show later in Section 6.6.6).

Table 6-8. Results of grouping algorithms for general processor architectures.

8(a) Linux L7-filter (70 Patterns)

Total DFA state limit	Groups	Total Number of States	Compilation Time (s)
3533	12	3371	5.602
4000	10	3753	7.335
10000	5	7280	37.928
32000	3	25215	49.976

8(b) Payload patterns from Bro (223 Patterns)

Composite DFA state limit	Groups	Total Number of States	Compilation Time (s)
5221	6	4697	1050
8000	4	6854	1030

For Bro's payload pattern set, the grouping algorithm can group more patterns into one group. Similar to the previous case, we start from 540, which is the largest individual DFA size. As Table 6-7(b) shows, we can group 222 patterns into 11 groups in this baseline case. As the DFA state limit increases, the number of groups decreases down to 4.

Table 6-8 demonstrates that the grouping algorithm for the general processor architectures can effectively reduce the number of groups generated as the memory limit imposed on the algorithm is increased. In addition, the total number of DFA states is close to the memory limit, showing the algorithm can fully utilize the memory allocated to the packet scanner. Note that we start the memory limit at 3533 DFA states for Linux L7-filter, which is the total number of the states of individual DFAs. This is the baseline of memory requirement without applying our algorithm for shared memory architectures. Some patterns can be grouped together without any extra memory usage because they do not interact with each other. In fact, we can group 70 patterns into 12 groups with no extra memory usage. When we increase the memory limit, more patterns can be grouped together because there is room for the DFAs to record the pattern interactions. As shown in the Table 6-8, we can decrease

the total number of groups to 3 when the state limit is 32000. Similarly to the Linux L7-filter, we start with 5221 DFA states for Bro payload set, which is the sum of 233 individual DFAs. Even without extra memory, the number of groups can be decreased from 233 to 6.

Beyond the effectiveness, Table 6-7 and Table 6-8 also present the running time of the proposed grouping algorithms. This overhead is a one-time cost. In networking environments, the packet content scanner operates continuously until there are new patterns to be inserted to the system. As patterns in the Linux L7-filter or the Bro system do not change frequently, the occasional overhead of several minutes associated with recomputation of groupings is affordable. The reason that these patterns do not change frequently is that the Linux L7-filter is for protocol detection and usually new protocols are not a common event. The intrusion detection systems like Bro and SNORT introduce new rules more often than Linux L7-filter, but still not on daily basis. It's on the average of once every ten days. A new pattern does not trigger a regroup of all patterns. We can just compute its pairwise interactions with existing patterns and pick a group that yields the fewest total interactions. This type of incremental update averages less than 1 second on the Bro payload pattern set using PCs with 3.4 GHz CPU and 3.7 GB memory.

6.6.6 Speed Comparison

We compare the proposed DFA-based algorithms with the state-of-the-art NFA-based regular expression matching algorithm. Both L7-filter and SNORT systems use this NFA-based library. We also compare it with the DFA-based repeated scan approach generated by Flex [104]. The results are summarized in Figure 6-9. The proposed DFA-based one pass scanner is 48 to 704 times faster than the NFA-based scanner. Compared to DFA-based repeated scan engine, our scanner yields a speed improvement of 11 to 42 times. Also note that although these dumps have dramatically different characteristics, our scanner provides similar

Table 6-9. Comparison of the different scanners.

		Throughputs (Mb/s)		Memory Consumption (KB)
		MIT dump	Berkeley dump	
Linux L-7 filter	NFA	1.0	3.4	1636
	DFA RP	16.3	34.6	7632
	DFA OP 3 groups	690.8	728.3	13596
Bro Http	NFA	30.4	56.1	1632
	DFA RP	117.2	83.2	1624
	DFA OP 1 group	1458.0	1612.8	4264
Bro Payload	NFA	5.8	14.8	1632
	DFA RP	17.1	25.6	7628
	DFA OP 4 groups	566.1	568.3	4312

NFA—NFA-based implementation
DFA RP – Flex generated DFA-based repeated scan engine
DFA OP – Our DFA one pass scanning engine

throughputs over these dumps because it scans each character only once. The other two approaches are subject to dramatic change in throughput (1.8 to 3.4 times) over these traces, because they need to do backtracking or repeated scans. Although the memory usage of our scanner is 2.6 to 8.4 times the NFA-based approach, the largest scanner we created (Linux L7-filter, 3 groups) uses 13.3 MB memory, which is well under the memory limit of most modern systems.

6.7 Summary

In this chapter, we considered the implementation of fast regular expression matching for packet payload scanning applications. Although regular expression matching has been a well studied problem, support for the specific class of patterns common in networking applications is new. We showed that the traditional NFA-based approaches are slow and naïve DFA implementations can have exponentially growing memory costs in the worst case. In this chapter, we analyzed the computational and storage cost of building individual DFAs

for matching regular expressions, and identify the structural characteristics of the regular expressions in networking applications that lead to exponential growth of DFAs. Based on this analysis, we proposed two rewrite rules for specific regular expressions. The rewritten rules can dramatically reduce the size of resulting DFAs, making them small enough to fit in memory. While we do not claim to handle all possible cases of dramatic DFA growth (in fact the worse case cannot be improved), rewrite rules do cover those patterns present in common payload scanning rulesets like SNORT and Bro, thus making fast DFA-based pattern matching feasible for today's payload scanning applications. It is possible that a new type of attack also generates signatures of large DFAs. For those cases, unfortunately, we need to study the signature structures before we can rewrite them. Besides rewriting, we presented a scheme that selectively groups patterns together to further accelerate the matching process.

The proposed DFA-based implementation is 2 to 3 orders of magnitude faster than the widely used NFA implementation and 1 to 2 orders of magnitude faster than a commonly used DFA-based parser. Our grouping scheme can run on general processor architectures, where the DFA for one group corresponds to one process or thread, as well as to multi-core architectures, where groups of patterns can be processed in parallel using independent processing units. In the future, it would be an interesting study to apply different DFA compression techniques and explore tradeoffs between the overhead of compression and the savings in memory usage.

We have finished all the technical chapters. Chapters 3 and 4 provided high speed multi-match classification algorithms for packet header processing. Chapter 5 and this chapter proposed schemes for pattern matching on packet payload, with Chapter 5 focusing on fixed patterns and this on the more general regular expressions. These algorithms together make high speed deep packet inspection possible. We will conclude this thesis in the next chapter and also discuss future directions.

7 Concluding Remarks and Future Work

This chapter concludes the dissertation by summarizing the major contributions of the thesis and suggesting the key direction for future work.

7.1 Summary of thesis work

In this dissertation, we designed high speed packet processing algorithms that function on the entire packet including both the header and the payload. These algorithms can enable new services such as network intrusion detection. To keep up with high speed packet processing in existing networks, our algorithms are optimized for new technologies such as Ternary Content Addressable Memory (TCAM) and multi-core processors.

For packet header processing, in Chapters 3 and 4, we proposed a multi-match packet classification scheme that intelligently processes packet header information. For packet payload processing, we developed schemes to identify fixed string patterns in Chapter 5 and complex regular expressions in Chapter 6. These techniques form a cohesive architecture that can perform gigabit rate packet scanning against thousands of sophisticated patterns. We summarize these techniques in detail below.

7.1.1 Multi-match Packet Classification

In Chapter 3, we developed a TCAM-based approach called Geometric Intersection Method that can report multi-match classification results with just two memory lookups: one TCAM lookup and one SRAM lookup. In addition, our negation removing scheme can efficiently map filters into the TCAMs, which can save 95% of TCAM memory compared to the straightforward solution.

The Geometric Intersection Method is fast, but it may not be power efficient because it can generate many intersection filters. To address this problem, we developed scheme called the Set Splitting Algorithm (SSA) to split filters into multiple groups. Through filter splitting, SSA guarantees a reduction in the number of intersection filters by 50% when a filter set is split into two. In addition, each filter in the TCAM is accessed once per packet, further reducing power consumption. Compared to previously published schemes, SSA saves 75% to 95% of power consumption.

7.1.2 Fixed String Matching with TCAM

For fixed string matching, we adopted TCAM because of its fast speed in parallel search. In Chapter 5, we developed TCAM-based algorithms for detecting long patterns, correlated patterns, and patterns with negation. Our simulation results showed that our scheme can handle patterns with dramatic variation in size, and can perform Gigabit rate pattern matching on both the SNORT and ClamAV virus database. Our scheme is also applicable to other Layer 7 pattern matching problems, for example, applications like HTTP load balancing and email SPAM filtering.

7.1.3 Fast Regular Expression Matching

In Chapter 6, we proposed fast regular expression matching scheme using general processors. We studied the typical patterns used in the networking applications. We showed that some pattern generate exponential size DFA when using traditional methods. To address this, we proposed rewrite techniques that can effectively reduce the size of. In addition, we developed grouping algorithms that compile a large set of regular expressions into a small number of DFA, which dramatically improves the regular expression matching speed without significantly increasing memory usage. We showed through simulation results that our implementation achieved one to three orders of magnitude speedup over the state-of-the-art

NFA-based implementation. Our scheme can be applied to both general processor-based and multi-core-based architectures. In addition, it can also be extended to FPGA and ASIC-based platforms.

7.2 Future Directions

Our work to date has demonstrated that Gigabit rate deep packet inspection can be achieved by combining well-designed algorithms and high performance hardware technologies. There are several interesting directions for future work that one could pursue based on the work presented in this dissertation. The first two are related to making the deep packet inspection system more powerful and the third one is on the application of our proposed schemes to a problem unrelated to detecting attackers.

1) Detecting attacks in real time without signatures: This dissertation is based on the assumption that virus and worm signatures are known in advance. We rely on third parties to provide these. Disseminating new signatures quickly after the worm breaks out is also a challenging task. This is because worm breakouts, such as the Slammer worm devastation, can consume a large portion of the Internet bandwidth and thus cause congestion [105]. To address this problem, we need algorithms to detect suspicious traffic *without using signatures* and limit the rate of suspicious traffic *before obtaining the signatures*, so as to provide better Quality of Service (QoS) for more important traffic, including routing information and signature dissemination. By limiting the rate of suspicious traffic, we can slow down the propagation of worms, which can buy us more time to filter them more precisely *with attack signatures*.

2) Coordinated attack detection architecture. There are thousands of routers in the Internet. It is highly inefficient to deploy the same set of signatures at every router. Because a packet travels through multiple routers, it also makes no sense to do redundant searches for

the same groups of signatures at different routers. To build a more efficient and coordinated packet scanning architecture, we need an intelligent scheme to deploy selected sets of signatures at different routers. The scheme must guarantee that a packet will be checked by the complete pattern set, no matter which path it chooses to travel in the Internet. This scheme can also help decrease per packet computation costs at routers and hence can potentially increase the pattern matching speed.

The above directions for future work are based on devising techniques to make the deep packet inspection systems easier to deploy in today's Internet. Next we point out a direction that is based on using the deep packet inspection in other high speed data processing applications.

3) High speed data processing such as XML message processing and biomedical data processing: XML message dissemination and bioinformatics systems, similar to network applications, generate a huge amount of data and require high speed pattern processing capabilities to search and locate desired information. The XML query processing language XPath is similar to regular expressions and we believe that our regular expression scheme can be applied to signatures expressed in XPath.

Some of the gene search algorithms such as BLAST [106] require pattern searching. We believe that our high speed fixed string matching solutions for network applications can be extended to apply to this area. However, our scheme needs to be modified as many gene searching algorithms aims to identify approximate patterns, rather than exact strings.

Appendix A

Claim in Section 3.4: If E_i is the first superset of x ($x \subset E_i$) in E, we can add x before E_i according to requirement (3) and bypass all the rules after E_i.

Proof: For any rule E_j after E_i, there could be four cases. We will study it one by one and show why we can bypass all of them.

First, we can bypass any rule E_j that is disjoint with x, according to requirement (1). Second, it is impossible that $E_j \subset x$. If so, $E_j \subset x \subset E_i$, which contradicts with requirement (2).

Third, If $x \subset E_j$, E_j must also be a superset of E_i. Otherwise, the intersection of E_j and E_i must be a superset of x as well and it must be presented before E_i, according to requirement (4). This contradicts with the assumption that E_i is the first superset of x in E. Therefore, $E_i \subset E_j$ and we have $M_j \subset M_i$ according to requirement (2). In this case, we don't need to process E_j since we can extract all the information from M_i.

Fourth case, if E_j intersects with x and suppose $z = E_j \cap x$, then z must have appeared before E_i. This is because E_j must intersect with E_i as well since E_i is a superset of x. Let $E_k = E_i \cap E_j$, according to requirement (4), $k < i$. In addition, $z = E_j \cap x = E_j \cap x \cap E_i = E_k \cap x$, because $x \subset E_i$. Therefore, we must have generated z when processing E_k which is before E_i. This meets the requirement (4), so we can bypass E_j.

Hence, all the rules after E_i are either exclusive to x, or their intersections have already been included, so we can skip all those rules.

Appendix B

Theorem 1: Pattern *P1 "^A[A-Z]{j}"* is equivalent to the original pattern *P2 "^A+[A-Z]{j}"* for detecting non-overlapping shortest string.

Proof: We prove this theorem through the following Claims 1 and 2.

Claim 1: Any input matching *P1* must match *P2* as well, and the shortest matched string *S1* for *P1* is the same as the shortest matched sting *S2* for *P2*.

Proof of Claim 1: For any input matching *P1*, it must match pattern *P2* because we can use *"\s"* to match *"\s+"* and the remaining *j* characters as [*A-Z*]{*j*}. Next we prove their matched string *S1* and *S2* are identical. For *P1,* there is only one way of selecting *S1* and its length is *j+1*. There maybe multiple ways of selecting *S2* (same start position, overlapping strings), with length from *j+1* to infinity. If we pick the shortest match, its length would also be *j+1*. In addition, *S1* and *S2* must start from the beginning of the input due to the requirement of ^. Given that they have the same length, *S1* and S2 must be identical.

Claim 2: Any input matching *P2* must match *P1* as well, and both patterns report matching of the same shortest string.

Proof of Claim 2: For any input matching *P2* "*^A+[A-Z]{j}", it must have x "A"s (x >= 1)* matched as "*^A+", y "A"s* and *z [A-Z]* characters (starting from a none *"A")* matched as "[*A-Z*]{*j*}" *(y >= 0, z >= 0, y+z = j).* This input must match *P1 "^A[A-Z]{j}"* because the input have *x-1+y+z >= j [A-Z]* characters after the first *A*. Similar to (1), the shortest matched strings are the same.

Since pattern starts with ^, *P1* and *P2* report at most one match for one line. Given Cliams 1 and 2, *P1* and *P2* report the same results for any input, hence they are equivalent. □

Theorem 2. Patter *P1* ".**AB[A-Z]{j}*" can be rewritten as pattern *P2* "*([^A]|A[^B]|AB[A-Z]{j-1}[^(A-Z)])*AB[A-Z]{j}*". These two patterns are equivalent for detecting non-overlapping strings.

Proof: It is trivial that these two patterns are equivalent when the input does not contain *AB\s* because none of them match the input. It is also trivial if the input only contains one *AB\s*. Next, we prove the case where we have multiple *ABs* without *[^(A-Z)]* in between and they are within *j* bytes to the first *AB* through Claims 3, 4, and 5.

Claim 3: Any input not matching *P2* does not match *P1* either.

Proof of Claim 3: Since the input does not match *P2*, there must be a *[^(A-Z)]* character within the next *j* bytes of the first *AB*, this character must also be located within *j* bytes to the following *ABs*. Hence, *P1* will not be matched either.

Claim 4: Any input matching *P2* must match *P1*. *P2* and *P1* generate matching results at the same position.

Proof of Claim 4: For any input matching *P2,* it must report matching result at *j* positions after the first *AB*. If there is no *[^(A-Z)]* character within the next *j* bytes to one of the *ABs*, then there will not be *[^(A-Z)]* within the next *j* bytes to the first *AB* because there is no *[^(A-Z)]* in between of these *ABs*. Therefore, the match result of *P1* will be generated *j* bytes after the first *AB* as well. Hence, *S1* and *S2* are the same.

Claim 5: *P1* and *P2* report the same number of matches.

Proof of Claim 5: When there are multiple *ABs* without *[^(A-Z)]* between them and they are within *j* bytes to the first *AB*. *P1* would only report one match, because these *ABs* are within *j* bytes and their matching strings overlap with each other. *P2* would report one match too. Hence, *P1* and *P2* report the same number of matches. If there are multiple non-overlapping patterns in the input (*ABs* are at least *j* apart or with *[^(A-Z)]* in between), *P1* and *P2* still

report the same number of matches because we can divide the input to segments, where only one match is reported in one segment.

Given (1), (2) and (3), for any input, patterns P1 and P2 report the same matching results and hence they are equivalent. □

Reference:

[1] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "The Spread of the Sapphire/Slammer Worm." http://www.cs.berkeley.edu/~nweaver/sapphire/.

[2] R. Lemos, "Counting the Cost of Slammer (CNET News.com)." http://news.zdnet.com/2100-3513_22-982955.html?tag=nl.

[3] R. Lemos, "'Slammer' Attacks May Become Way of Life for Net." http://news.com.com/Damage+control/2009-1001_3-983540.html.

[4] R. Lemos, "Slammer Report: More Headaches." http://news.zdnet.com/2100-1009_22-983736.html.

[5] S. Cowley and M. Williams, "Slammer Worm Slaps Net Down, But Not Out (IDG News)." http://www.pcworld.com/news/article/0,aid,108988,00.asp.

[6] Wikipedia.org, "Mydoom." http://en.wikipedia.org/wiki/MyDoom.

[7] mi2g, "MyDoom Becomes Most Damaging Malware as SCO is Paralysed." http://www.mi2g.com/cgi/mi2g/frameset.php?pageid=http%3A//www.mi2g.com/cgi/mi2g/press/010204.php.

[8] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, 1984.

[9] "ISC Domain Survey: Number of Internet Hosts." http://www.isc.org/index.pl?/ops/ds/host-count-history.php.

[10] "SNORT Network Intrusion Detection System." http://www.snort.org.

[11] "Sun Managed Security Services." http://www.sun.com/service/managedservices/MSSequipment.pdf.

[12] "Sun Managed Security Services: Multidimensional Defense-in-Depth," A Sun Microsystems White Paper. http://www.sun.com/service/managedservices/Man_Sec_WP11.15.pdf.

[13] H. A. Kim and B. Karp, "Autograph: Toward Automated, Distributed Worm Signature Detection," *Proc. 13th Usenix Security Symposium*, August 2004.

[14] C. Kruegel, D. M. E. Kirda, W. Robertson, and G. Vigna, "Polymorphic Worm Detection Using Structural Information of Executables," *Proc. 8th InternationalSymposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[15] Z. Liang and R. Sekar, "Automatic generation of buffer overflow attack signatures: An approach based on program behavior models," *Proc. ACSAC*, 2005.

[16] "Bro Intrusion Detection System." http://bro-ids.org/Overview.html.

[17] "Clam AntiVirus signature database." www.clamav.net.

[18] Richard E White, D. M. James, and I. Buchholz (Wheeling, "Packet reassembly method and apparatus." US, 1995.

[19] M. Necker, D. Contis, and D. Schimmel, "TCP-Stream Reassembly and State Tracking in Hardware," *Proc. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002.

[20] S. Dharmapurikar and V. Paxson, "Robust TCP Stream Reassembly In the Presence of Adversaries," *Proc. 14th USENIX Security Symposium*, 2005.

[21] H. Zimmermann, "OSI Reference Model--The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications*, 1980.

[22] J. Levandoski, E. Sommer, and M. Strait, "Application Layer Packet Classifier for Linux." http://l7-filter.sourceforge.net/.

[23] B. W. Watson, "The Performance of Single-keyword and Multiple-keyword Pattern Matching Algorithms," *Computing Science Note, Eindhoven University of Technology*, 1994.

[24] R. Sommer and V. Paxson, "Enhancing Byte-Level Network Intrusion Detection Signatures with Context," *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2003.

[25] C. J. Coit, S. Staniford, and J. McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of SNORT," *DARPA Information Survivability Conference and Exposi-tion (DISCEX II'01)*, 2001.

[26] M. Fish and G. Varghese, "Fast Content-Based Packet Handling for Intrusion Detection," *UCSD technical report CS2001-0670,*, 2001.

[27] "CERT® Advisory CA-2001-19 "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL." http://www.cert.org/advisories/CA-2001-19.html.

[28] M. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. Campbell, "Directions in Packet Classification for Network Processors," *Proc. NP2 Workshop*, February 2003.

[29] M. H. O. a. A. F. Stappen, "Range searching and point location among fat objects," *European Symposium on Algorithms*, pp. 240-253, 1994.

[30] P. Gupta and N. McKeown, "Packet classification on multiple fields," *Proc. SIGCOMM*, August 1999.

[31] D.R.Morrison, "PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric," *Journal of the ACM (JACM) archive*, vol. 15, 1968.

[32] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and Scalable Layer Four Switching," *Proc. Sigcomm*, Spetember 1998.

[33] M. M. Buddhikot, S. Suri, and M. Waldvogel, "Space Decomposition Techniques for Fast Layer-4 Switching," *Protocols for High Speed Network*, vol. 66, 1999.

[34] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," *Proc. ACM Sigcomm*, 1999.

[35] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," *Proc. Hot Interconnects*, August, 1999.

[36] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet Classification Using Multidimensional Cutting," *Proc. Sigcomm*, August 2003.

[37] C. Partridge, "Locality and route caches," *Proc. NSF Workshop on Internet Statistics Measurement and Analysis*, 1996.

[38] P. Newman, G. Minshall, and L. Huston, "IP switching and gigabit routers," in *IEEE Communications Magazine*, 1997.

[39] M. Gokhale, D. Dubois, A. Dubois, M.Boorman, S. Poole, and V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection Technology," *Proc. FPL, Lecture Notes in Computer Scienc,e*, 2002.

[40] "Overview of the Fast Ethernet and Gigabit Ethernet SPAs," *Cisco Systems*.

[41] "PM2329 Classipi Network Classification Processor Datasheet," in *Cypress cooperation*, 2001.

[42] "Security Features on Cisco 1800, 2800 and 3800 Integrated Services Routers," 2004.

[43] "5512GLQ TCAM from NetLogic Microsystems."

[44] D. E. Taylor, "Survey Taxonomy of Packet Classification Techniques."

[45] "Measurement & Operations Analysis Team from the National Library for Applied Network Research (NLANR) project," 2001.

[46] D. E. Taylor, "Survey Taxonomy of Packet Classification Techniques," *Proc. Tech Report, WUCSE-2004-24*, May 2003.

[47] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: Power-Efficient TCAMs for Forwarding Engines," *INFOCOM*, March 2003.

[48] "FPGA." http://en.wikipedia.org/wiki/Fpga.

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

[50] J. W. Lockwood, J. Moscola, D. Reddick, M. Kulig, and T. Brooks, "Application of hardware accelerated extensible network nodes for internet worm and virus protection," *International Working Conference on Active Networks (IWAN)*, 2003.

[51] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun, "The Stanford Hydra CMP," in *IEEE MICRO Magazine*, 2000.

[52] "Intel® IXP2800 network processor." http://www.intel.com/design/network/products/npfamily/ixp2800.htm.

[53] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM J. RES. & DEV.*, vol. 49, JULY/SEPTEMBER 2005.

[54] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," *Proc. ACM SIGCOMM*, 2006.

[55] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for Advanced Packet Classification with Ternary CAMs," *Proc. ACM Sigcomm*, 2005.

[56] S. Ramabhadran and G. Varghese, "Efficient Implementation of a Statistics Counter Architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31.

[57] F. Baboescu, S. Singh, and G. Varghese, "Packet Classification for Core Routers: Is there an alternatives to CAMs?" *Proc. IEEE Infocom*, 2003.

[58] J. Bolaria and L. Gwennap, "A Guide to Search Engines and Networking Memory." http://www.linleygroup.com/Reports/memory_guide.html.

[59] M. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. Campbell, "Directions in Packet Classification for Network Processors," *Proc. NP2 Workshop*, Feburary 2003.

[60] A. Hari, S. Suri, and G. Parulkar, "Detecting and Resolving Packet Filter Conflicts," *Proc. INFOCOM*, 2000.

[61] H. Liu, "Reducing Routing Table Size Using Ternary-CAM," *Proc. Hot Interconnects*, 2001.

[62] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and Scalable Layer Four Switching," *Proc. Sigcomm*, September 1998.

[63] V. Srinivasan and G. Varghese, "Faster IP lookups using controlled prefix expansion," *Proc. ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, 1998.

[64] S. Iyer, R. R. Kompella, and A. Shelat, "ClassiPI: An Architecture for Fast and Flexible Packet Classification," *IEEE Network Spec. Issue*, vol. 15, 2001.

[65] M. Kobayashi, T. Murase, and A. Kuriyama, "A longest prefix match search engine for multi-gigabit ip processing," *Proc. ICC 2000*, June 2000.

[66] E. Spitznagel, D. Taylor, and J. Turner, "Packet Classification Using Extended TCAMs," *Proc. IEEE International Conference on Network Protocols (ICNP)*, 2003.

[67] F. Yu and R. H. Katz, "Efficient Multi-Match Packet Classification with TCAM," *Hot Interconnects*, August, 2004.

[68] R. Panigrahy and S. Sharma, "Reducing TCAM Power Consumption and Increasing Throughput," *Proc. Hot Interconnects*, 2001.

[69] K. Zheng, H. Che, Z. Wang, and B. Liu, "an ultra high throughput and power efficient TCAM based IP lookup engine," *Proc. IEEE Infocom*, 2004.

[70] H. Song and J. W. Lockwood, "Efficient packet classification for network intrusion detection using FPGA," *Proc. International Symposium on Field Programmable Gate Arrays (FPGA)*, Monterey, USA, 2005.

[71] D. S. Johnson, "Approximation algorithms for combinatorial problems," *Proc. the fifth annual ACM symposium on Theory of computing*, 1973.

[72] P. Crescenzi and V. Kann, "A compendium of NP optimization problems." http://www.nada.kth.se/~viggo/wwwcompendium/node145.html.

[73] G. Andersson and L. Engebretsen, "Better Approximation Algorithms and Tighter Analysis for SET SPLITTING and NOT-ALL-EQUAL SAT," *Proc. technical reports, ECCCTR: Electronic Colloquium on Computational Complexity*, 1998.

[74] M. X. Goemans and D. P. Williamson, "Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming," *J. Assoc. Comput. Mach*, vol. 42, pp. 1115--1145, 1995.

[75] Z. Kerekes, "History of SPARC systems," SPARC Product Directory, 1997.

[76] M. Hidell, P. Sjödin, and O. Hagsand, "Router Architectures," *Tutorial at Networking 2004*.

[77] M. Kobayashi, T. Murase, and A. Kuriyama, "A longest prefix match search engine for multi-gigabit ip processing," *Proc. International Conference on Communications (ICC 2000)*.

[78] E. Nordmark, "Contribution of packet sizes to packet and byte volumes." http://www.nlanr.net/NA/Learn/packetsizes.html.

[79] "Intel Pentium 4 Processor." http://www.intel.com/products/processor/pentium4/index.htm.

[80] "Sync SRAM Products," Cypress Coperation,. http://www.cypress.com/portal/server.pt?space=CommunityPage&control=SetCommunity&CommunityID=209&PageID=215&gid=5&fid=39&category=All&showall=false.

[81] D. E. Knuth, J. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing*, 1997.

[82] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Communications of ACM*, vol. 20, 1997.

[83] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of ACM*, vol. 18, 1975.

[84] B. Commentz-Walter, "A String Matching Algorithm Fast on the Average," *ICALP, LNCS*, vol. 6, 1979.

[85] Z. K. Baker and V. K. Prasanna, "A methodology for synthesis of efficient intrusion detection systems on FPGAs.," *Proc. Field-Programmable Custom Computing Machines (FCCM)*, 2004.

[86] S. Dharmapurikar, M. Attig, and J. Lockwood, "Deep packet inspection using parallel bloom filters," *IEEE Micro*, 2004.

[87] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection," *Proc. The international conference on Field-Programmable Logic and its Applications (FPL)*, 2002.

[88] J. Mudigonda, H. M., and R. Yavatkar, "Overcoming the memory wall in packet processing: hammers or ladders," *Proc. Symposium On Architecture For Networking And Communications Systems*, 2005.

[89] "Snot Packet Generator," Discussion on SecurityFocus IDS Mailing List. http://www.securityfocus.com/ids.

[90] J.-S. Sung, e.-M. Kang, Y. Lee, T.-G. Kwon, and B.-T. Kim, "A multi-gigabit rate deep packet inspection algorithm using TCAM," *Proc. IEEE Global Telecommunications Conference GLOBECOM*, 2005.

[91] "Europe Surpasses North America In Instant Messenger Users," comScore Networks Study. http://www.comscore.com/press/release.asp?press=800.

[92] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," *Proc. 32nd Annual International Symposium on Computer Architecture (LISA)*, Madison, Wisconsin, 2005.

[93] Y. Cho and W. Mangione-Smith, "Deep packet filter with dedicated logic and read only memories," *Proc. Symposium on Field-Programmable Custom Computing Machines*, 2004.

[94] Z. K. Baker and V. K. Prasanna, "Time and area efficient pattern matching on FPGAs," *Proc. International Symposium on Field Programmable Gate Arrays (FPGA)*, 2004.

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

[96] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit Rate Packet Pattern Matching with TCAM," *Proc. ICNP*, Berlin, Germany, October, 2004.

[97] Y. H. Cho and W. H. MangioneSmith, "A Pattern Matching Coprocessor for Network Security," *Proc. DAC*, 2005.

[99] Venkat, "Yahoo Messenger Protocol." http://www.venkydude.com/articles/yahoo.htm.

[100] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Second ed: Addison Wesley, 2001.

[101] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML Streams with Deterministic Automata and Stream Indexes," *ACM TODS*, vol. 29, 2004.

[102] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer, "Path Sharing and Predicate Evaluation for High-Performance XML Filtering," *ACM TODS, 28(4)*, 2003.

[103] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, J. Turner, "Algorithms to accelerate Multiple Regular Expression Matching for Deep Packet Inspection," *ACM Sigcomm, 2006*.

[104] V. Paxson and etc., "Flex: A fast scanner generator." http://www.gnu.org/software/flex/.

[105] M. Lad, X. Zhao, B. Zhang, D. Massey, and L. Zhang, "Analysis of BGP update surge during Slammer worm attack," *Proc. International Workshop on Distributed Computing*, 2003.

[106] S. F. Altschul, W. Gish, Miller W., M. E.W., and D. J. Lipman, "Basic local alignment search tool," *J. Mol. Biol.*, 1990.

[107] W. Eatherton, "The Push of Network Processing to the Top of the Pyramid", ANCS 2005.