Stony Brook University

Efficient Techniques for Fast Packet Classification

A Dissertation Presented
by
Alok Tongaonkar

to
The Graduate School
in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy
in
Computer Science
Stony Brook University

August 2009

Stony Brook University

The Graduate School

Alok Tongaonkar

We, the dissertation committee for the above candidate for
the degree of Doctor of Philosophy,
hereby recommend acceptance of this dissertation.

Professor R. Sekar, (Advisor)
Computer Science Department, Stony Brook University

Professor I. V. Ramakrishnan, (Chairman)
Computer Science Department, Stony Brook University

Professor Robert Johnson, (Committee Member)
Computer Science Department, Stony Brook University

Professor Nitesh Saxena, (External Committee Member)
Computer Science and Engineering Department, Polytechnic Institute of NYU

This dissertation is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Efficient Techniques for Fast Packet Classification

by

Alok Tongaonkar

Doctor of Philosophy

in

Computer Science

Stony Brook University

2009

Rule-based packet classification plays a central role in network intrusion detection systems, firewalls, network monitoring and access-control systems. To enhance performance, these rules are typically compiled into a *matching automaton* that can quickly identify the subset of rules that are applicable to a given network packet. The principal metrics in the design of such an automaton are its size and the time taken to match packets at runtime. Previous techniques for this problem either suffered from high space overheads (i.e., automata could be exponential in the number of rules), or matching time that increased quickly with the number of rules. In contrast, we present a new technique that constructs polynomial size automata. Moreover, we show that the matching time of our automata is insensitive to the number of rules. The key idea in our approach is that of decomposing and reordering the tests contained

in the rules so that the result of performing a test can be utilized on behalf of many rules. Our experiments demonstrate dramatic reductions in space requirements over previous techniques, as well as significant improvements in matching speed. Our technique can uniformly handle prioritized and unprioritized rules, and support single-match as well as multi-match classification.

To my parents,

my sister Meghana,

my brother-in-law Nitin,

and my wonderful nephew Aryan.

Contents

List of Figures

Acknowledgments

In poverty and other misfortunes of life, true friends are a sure refuge. The young they keep out of mischief; to the old they are a comfort and aid in their weakness, and those in the prime of life they incite to noble deeds. — Aristotle.

Working on my PhD dissertation has been a wonderful journey. I am grateful to take this opportunity to sincerely thank all those who helped me on the way. First and foremost, I would like thank my advisor, Prof. R. Sekar, for his constant support, invaluable guidance and infinite patience. His work ethic and constant endeavor to achieve perfection have been a great source of inspiration.

I wish to extend my sincere thanks to Prof. I.V. Ramakrishnan, Prof. Robert Johnson, and Prof. Nitesh Saxena for consenting to be on my defense committee and offering invaluable suggestions to improve this report. I am thankful for what I learnt, inside and outside the classroom, from Prof. Michael Bender, Prof. C.R. Ramakrishnan, Prof. Tzi-cker Chieuh, Prof. Qin Lv, and Prof. Scott Stoller.

I would like to thank my friends: Akshay Athalye, Vishnu Navda, Amit Purohit, Sumukh Shevde, Vaishali and Ibrahim Muckra, Swetha and Mohan Reddy, Shruti and Salil Gokhale, Uttara and Gaurav Gothoskar, Sandeep Bhatkar, Namit Joshi, and Varun Puranik for being my family away from home; Diptikalyan Saha and Girish Ramakrishnan for being great roommates; Niranjan Inamdar, Sreenaath Vasudevan, Mayur Mahajan, Manuel Rivera, Vishal Chowdhary, T. V. Lakshmi Kumar, and Lohit Vijayrenu for working tirelessly on projects with me; Kiran Kumar Reddy, Wei Xu, Zhenkai Liang, and Weiqing Sun for providing much needed guidance; Varun Katta

and Mithun Iyer for being great sounding boards; Sandhya Menon, Namrata Godbole, Prachi Kaulgud, Srivani Narra, and Anupama Chandwani for providing many lighter moments that were a welcome relief from the humdrum of graduate life; Lorenzo Cavallaro and Yves Younan for bringing European touch to the lab; Pinkesh Zaveri, Gaurav Salia, Gopalan Sivathanu, Michelle Carmelo, and Neeti Gore for being there when I needed them; Abhiraj Butala, Arvind Ayyengar, Bhuvan Mittal, Ashish Misra, and Prachi Deshmukh for seeing me over the finishing line; and countless other friends in Stony Brook.

I would also like to thank my friends who have encouraged and inspired me even before I started this journey: Rudhir Patil, Nirjhar Goel, Shruti Singh, Sonali Singh, Mangesh Edke, Milap Paun, Gulzar Wadiwala, Suyog Lokhande, Omkiran Sharma, Trupti Suryavanshi, Ashutosh Deshpande, and Niranjan Potnis.

I have been fortunate to work with mentors who were really passionate about their work: Morton Swimmer and Sailesh Kumar; managers who were very approachable and friendly: Andreas Wespi and Pere Monclus; and Fyodor, Andrea Baldini, and Valentina Alaria who gave me an opportunity to work on some really interesting projects. I wish I could have spent more time working on projects with all of them. I am also thankful to Karthick Iyer for providing a new challenge for me. The best part of my internships was the friendships that I formed there – the pool group in Zurich – Lukasz Juszczyk, Corrado Leita, Alex Mathey, Milton Yates, Karima Rehioui, and Michel Berard, and Domenico Ficara – in California – who was always ready for a dinner outing.

I would like to thank Brian Tria, Betty Knittweis, Kathy Germana, Cindy Scalzo, Chris Kalesis, and Mara Green who got me out of countless sticky administrative situations.

Finally, and most importantly, I would like to thank my parents, sister Meghana, brother-in-law Nitin Sadawarte, and my extended family without whose support this work would have been impossible.

This research is supported mainly by an ONR grant N000140110967 and in part by NSF grants CCR-0098154 and CCR-0208877.

CHAPTER 1

Introduction

The past few years have seen an explosive growth in the number of systems connected to networks. This has radically changed the way we perform many day-to-day tasks such as accessing our bank accounts, buying and selling, reading news or books, listening to music, watching live events, and staying in touch with people. Growth of computer networks has made it easier to perform these tasks. Businesses and governments are also taking advantage of the additional convenience and cost reductions brought about by this growth. They are becoming increasingly reliant on computer networks for their day-to-day operations.

The result of this reliance on networked systems is that security of these systems has become of paramount importance to our society. Security breaches can have consequences like huge financial losses. The problem is worsened by the fact that cyber crimes have become financially lucrative, leading attackers to use more and more resources to exploit vulnerabilities in networked systems. It is therefore crucial to employ various system and network security mechanisms that can thwart these attacks.

The two main goals of network security are (*i*) to protect data in transit, and (*ii*) to protect end host systems from unauthorized access. Mechanisms that use cryptographic techniques are commonly used to achieve the first goal. In this dissertation, we will focus on applications such as firewalls, signature-based network intrusion detection systems, and networking monitoring tools

that try to achieve the latter goal. These applications form the first-line of defense for many networked systems. They inspect network packets and try to identify packets that can lead to unauthorized access. They are typically rule-based systems, where the rules are used to specify how to process network packets:

- *Firewalls* and *access control systems* are used to control access between hosts on different networks. They are deployed on the edge of a private network that needs to be protected. The hosts on the network being protected are considered as *inside* hosts. All other hosts are considered as belonging to the *outside* network. These systems permit or deny incoming (or outgoing) network packets based on the conditions specified in firewall or access-control rules. These rules specify which hosts and/or networks are allowed access to the services provided by inside hosts and which outside services are allowed access by inside hosts. These systems look for the first matching rule in a linearly ordered rule set.

- *Network intrusion detection systems* (NIDS) are used to detect attacks over the network. Network intrusion detection system rules define suspicious activity using patterns that are observed in network packets involved in known attacks. To avoid missing any potential attack, NIDS need to match network packets against unordered rule sets and identify all the matching rules. NIDS are deployed at key points of the network like the edge to protect the network infrastructure.

- *Network monitoring applications* are deployed on network devices to selectively monitor, record, or analyze network packets belonging to a particular traffic. These applications use rules to specify conditions on the packets of interest. Typically, they are concerned with *packet filtering*, i.e., identifying if any of the conditions is satisfied rather than identifying the matching rules.

The key challenge in deploying these security mechanisms effectively in today's networks is their performance. Network speeds have increased to the

point where gigabit link rates are commonplace now at the edge and in the core of many networks. Firewalls, network intrusion detection systems, and networking monitoring systems need to operate at these line-rates to avoid dropping packets or missing unauthorized accesses. One of the key factors that determines the performance of these systems is the amount of processing time that they spend in identifying the matching rules for each network packet. Achieving acceptable performance for rule matching has become very challenging due to the increasing size and complexity of the typical rule sets used by these applications. Nowadays, network intrusion detection systems rule sets contain several thousand rules to deal with the rapid escalation in the number of new attacks. Similarly, the growth in network sizes and the number of applications supported has resulted in typical firewall rule set size to grow to several hundred rules.

In this dissertation, we present techniques for improving the performance of packet classification, which is the mechanism that inspects a network packet and determines how it is to be processed. In essence, given a set of rules $\{R_1, ..., R_n\}$, a *packet classifier* identifies the subset of rules that match the packet. These rules typically contain tests on packet header fields and patterns for matching against the packet payload. This latter operation is commonly referred to as *deep packet inspection* or *content-matching*. In the following sections we describe our approach for generating fast and scalable packet classifiers that can be used in a variety of applications.

1.1 Packet Classification

A naive technique for packet classification is that of sequentially matching each rule against an incoming packet. The performance of such a technique degrades linearly with the number of rules. Since the number of rules used in network intrusion detection systems and firewalls are typically large, this naive technique will not scale even to moderate speed networks.

The naive technique repeats computations involved in matching: in particular, a test that occurs in multiple rules is tested once on behalf of each rule.

This repetition can be avoided by building a finite-state automaton (referred to as *matching automaton*): the states of the automaton can be used to "remember" the tests already performed before reaching a state, and avoid repetitions. Transitions in the automata correspond to simple tests (e.g., equality or inequality checks) involving packet fields, and the final states indicate a match with one or more rules. Traditionally, finite-state automata have been used for deep packet inspection. In particular, finite-state automata can identify the matching strings or regular expression in a single pass over the payload.

A finite-state automaton can be either *deterministic* or *nondeterministic*. A deterministic automaton can identify all matching rules in a single scan of the input packet in time that is independent of the number of rules. Unfortunately, previous research has established an exponential lower bound on the size of deterministic automata [24], even in the simple case where we are restricted to equality checks with constants. Nondeterministic automata (also called as *backtracking* automata) do not suffer from this exponential blowup, but have the drawback that packet fields can be reexamined again and again. The matching times, in practice, can increase quickly with the number of rules for such automata. Previous packet classification techniques either used deterministic automata ([14]) or relied on nondeterministic automata ([18], [7], [4], [3]). Clearly, neither alternative is satisfying:

- Exponential blow-ups can't be tolerated since the number of rules can be large, e.g., several hundred to many thousands in the case of network intrusion detection systems and firewalls.

- Even a modest rate of increase in matching time with number of rules may not be acceptable in high-speed networks. For instance, consider a technique whose matching time increases at the rate of $\sqrt{N}$, where N is the number of rules. It slows down by a factor of 50 when the number of rules is increased to 2500.

1.1.1 Overview of Our Approach

Matching automata have been studied extensively in the context of term indexing and functional programming. Lot of works in these areas have focused on reducing the space and matching time requirements of such automata. Previous research has shown that both space and matching time of deterministic matching automata can be improved by designing a traversal order to suit the input rule set rather than using a predefined order [24]. Unfortunately, these results do not hold in the more general setting setting of packet classification, where disequalities and inequalities also need to be handled. Moreover, packet matching needs to support arbitrary bitmasking operations that further complicate designing such traversal order.

In this dissertation, we develop an algorithm for constructing classification automata that generalizes and applies the ideas developed in these areas. Our approach improves classification speed using a novel technique called *condition factorization* that breaks down tests involving packet fields in such a manner as to expose commonalities across different types of tests such as equality tests, inequality tests, tests involving bit-masking operations, etc. Moreover, in contrast with previous techniques, our algorithm guarantees a polynomial size automata in the worst-case. Our experimental results indicate that we construct classification automata that are tens to hundreds of times smaller than previous techniques while improving classification time substantially. Moreover, our experiments indicate an overall performance gain of 30%.

1.1.2 Contributions

The key contributions of our work are summarized below.

- Security applications have differing packet classification requirements. Firewalls need to identify the first matching rule from a linearly ordered rule set. Network intrusion detection systems look for all matching rules from unordered rule sets. Network monitoring may require only the ability to identify if a packet matches any of the rules. Previous research

efforts address one specific flavor of the matching problem. In contrast, we present a new packet classification technique that addresses *single-match* as well as *multi-match* classification, and supports *ordered* and *unordered* rules within a uniform framework.

- We develop the concept of *condition factorization* which is the core operation behind our algorithm. Condition factorization refers to decomposition and reordering of the tests contained in packet classification rules so that the result of performing a test can be utilized on behalf of as many rules as possible.

- We develop several techniques for selecting the order of tests to build space- and time-efficient automata. Our experiments show that a combination of these techniques is needed to achieve significant reduction in space and classification time of the automata.

  (a) We develop the notion of a *partitioning test* which makes the automaton size polynomial in the size of input rules if such tests are selected at every state. Typically, fields in packet headers that are used to identify higher layer protocols, are partitioning tests.

  (b) We present a new technique that guarantees polynomial space bounds (where the degree of the polynomial can be user-specified) by trading off some determinism. We point out that this theoretical possibility of nondeterminism wasn't observed in our experiments. Thus, our technique was able to guarantee quadratic worst-case space requirement, without incurring, in practice, the performance penalties associated with nondeterminism.

  (c) We develop the notion of *benign nondeterminism*, which enables the introduction of nondeterministic branches in the automaton *without any increase in matching times.* Use of benign nondeterminism can lead to dramatic reductions in automata size for certain rule sets.

  (d) To improve the matching times we develop the concept of *utility* of a test which captures the notion of how useful the test is in matching

the rule set. We try to pick tests with high utility values at each state to minimize the matching time of the automaton.

- We use a packet classification language that uses a strong type system similar to packet types [5] to ensure the safety of the matching code generated by our technique. This enables native code to be used for matching, resulting in very fast matching times, as opposed to previous techniques which use interpreters for matching their automata. The experimentally observed classification time remains virtually constant, regardless of the number of rules.

- We develop a metric for matching time that can be understood independent of the specifics of underlying hardware or software implementations to evaluate the performance of the classifiers generated by our technique.

- We generalize and extend these techniques to improve the performance of deep packet inspection as used in security applications like network intrusion detection systems.

1.2 Dissertation Organization

The rest of the dissertation is organized as follows. Part I describes our techniques for packet header matching. We formally define the packet classification problem in Chapter 2. We present our technique of condition factorization in Chapter 3. An algorithm for constructing packet classification automata using condition factorization is presented in Chapter 4. This algorithm is parameterized with respect to a function that selects tests to be performed at each automata state. Techniques for reducing size and matching time of automata are described in Chapter 5. Chapter 6 provides the details about how these techniques are implemented for the selection function. Chapter 7 presents experimental results. Chapters 8 in Part II discusses how our technique can be extended to handle deep packet inspection and shows the improvement in

overall system performance due to our technique. Related work is described in Chapter 9 followed by concluding remarks and future work in Chapter 10.

Part I

Packet Header Matching

CHAPTER 2

Background

Security applications like firewalls, network intrusion detection systems (NIDS), and network monitoring tools use rules of the form "*cond* $\longrightarrow$ *act*," where *act* specifies the action to be taken on a packet that matches the condition *cond*. Given a set of rules $\{R_1, ..., R_n\}$, and a packet p, a *packet classifier* identifies the rules, R_i, that match p.

In this chapter we present the packet classification problem. We focus on packet header matching in this part. We defer the discussion about deep packet inspection to Part II.

2.1 Preliminaries

In the rest of this dissertation, we are concerned only with the condition components of classification rules, which are referred to as *filters* henceforth. We associate a label with each filter to identify the corresponding rule. For simplifying the presentation, we will not consider content-matching initially, but we will show later on (in Chapter 8) how they can be encoded into packet filtering rules of the form discussed here.

Definition 2.1 (Tests, Filters and Priorities) *A **test** involves a variable x and one or two constants (denoted by c) and has one of the following forms.*

- Equality tests of the form $x = c$

- Equality tests with bitmasks of the form $x\&c_1 = c$
- Disequality tests of the form $x \neq c$
- Disequality tests with bitmasks of the form $x\&c_1 \neq c$
- Inequality tests of the form $x \leq c$ or $x \geq c$

*A **filter** F is a conjunction of tests. A set $\mathcal{F}$ of filters may be partially ordered by a priority relation. The priority of F is denoted as $Pri(F)$.*

An example of a filter, as defined above, is

$$(\texttt{dport} = \texttt{22}) \wedge (\texttt{sport} \leq \texttt{1024}) \wedge (\texttt{flags} \ \& \ \texttt{0xb} = \texttt{0x3})$$

We do not consider more complex conditions that do not satisfy the definition of a filter, e.g.,

$$(\texttt{sport} + \texttt{dport} < \texttt{1024}) \wedge (\texttt{sport} < \texttt{ttl}),$$

since they do not seem to arise in practice in our application domains (firewalls and network intrusion detection systems).

A filter F can be "applied" to a network packet p, denoted $F(p)$, by substituting variables, which denote the names of packet fields, with the corresponding values from p. We define the notion of matching based on whether the filter evaluates to *true* after this substitution. For example, consider a filter F_1 : ($\texttt{icmp\_type} = \texttt{ECHO}$). At packet classification time, for each input packet, the value of `icmp_type` field is obtained from that packet, and then the test is performed.

Definition 2.2 (Prioritized Matching) *For a set $\mathcal{F}$ of filters, we say that $F \in \mathcal{F}$ **matches a packet** p, denoted $M_{\mathcal{F}}(F, p)$, provided:*

- *$F(p)$ is true, and*
- *$F'(p)$is $false$, $\forall F' \in \mathcal{F}$ that have a strictly higher priority than F.*

*The **match set** of p, denoted $\mathcal{M}_{\mathcal{F}}(p)$ consists of all filters that match p, with the exception that among equal priority filters, at most one is retained in $\mathcal{M}_{\mathcal{F}}(p)$.*

Thus, a filter cannot match a packet unless matches with higher priority filters are ruled out. To illustrate matching, consider the following filter set $\mathcal{F}$:

- $F_1 : (\texttt{icmp\_type} = \texttt{ECHO})$
- $F_2 : (\texttt{icmp\_type} = \texttt{ECHO\_REPLY}) \wedge (\texttt{ttl} = 1)$
- $F_3 : (\texttt{ttl} = 1)$

Also consider an *icmp echo* packet p_1 and an *icmp echo reply* packet p_2, both having a *ttl* of 1.

- If these filters have incomparable priorities, then F_1 matches p_1, F_2 matches p_2, and F_3 matches both. As a result, $\mathcal{M}_{\mathcal{F}}(p_1) = \{F_1, F_3\}$ and $\mathcal{M}_{\mathcal{F}}(p_2) = \{F_2, F_3\}$
- If $Pri(F_1) > Pri(F_2) > Pri(F_3)$, then $\mathcal{M}_{\mathcal{F}}(p_1) = \{F_1\}$, and $\mathcal{M}_{\mathcal{F}}(p_2) = \{F_2\}$.
- If $Pri(F_3) > Pri(F_2) > Pri(F_1)$, then $\mathcal{M}_{\mathcal{F}}(p_1) = \mathcal{M}_{\mathcal{F}}(p_2) = \{F_3\}$.
- If $Pri(F_1) = Pri(F_3) > Pri(F_2)$, then $\mathcal{M}_{\mathcal{F}}(p_1)$ can either be $\{F_1\}$ or $\{F_3\}$, while $\mathcal{M}_{\mathcal{F}}(p_2) = \{F_3\}$.

These examples illustrate how various flavors of matching can be captured using priorities.

- *Packet-filtering* can be done by setting equal priorities for all filters. By virtue of the definition of match sets, this priority setting causes a match to be announced as soon as a match for any filter is identified.
- *Ordered matching,* as used in firewalls and access control lists can be done by assigning priorities that decrease monotonically with the rule number.

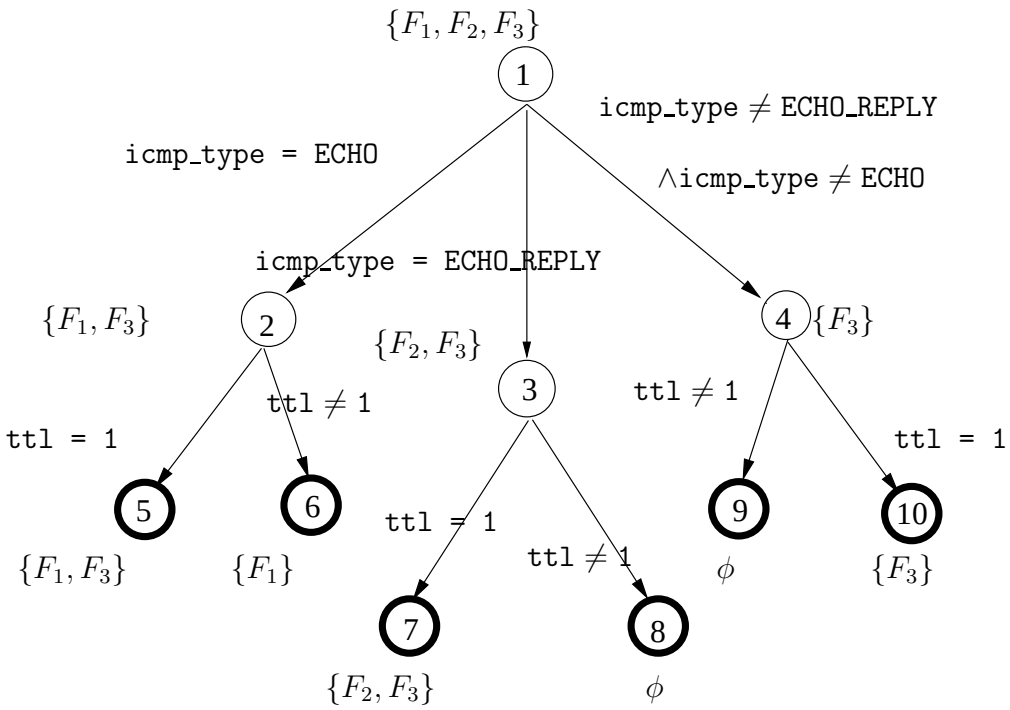

Figure 1: A deterministic classification automaton.

- *Multi-matching,* as used in network intrusion detection systems, can be solved by using incomparable priorities among filters.

2.2 Packet Classification Automata

In this section we describe *packet-classification automata* (also known as matching or classification automata). Examples of *packet-classification automata* (PCA) for the filter set in Section 2.1 with incomparable priorities is shown in Figures 1 and 2. Figure 1 shows a *deterministic automaton* (DCA), in which all of the transitions from any automaton state are mutually exclusive. A *nondeterministic automaton* (NCA) is shown in Figure 2, where the transitions may not be mutually exclusive. We make the following observations about the structure of classification automata:

- All but one of the transitions from each state are labeled with a *test* as defined above; the remaining (optional) transition, called an "other"

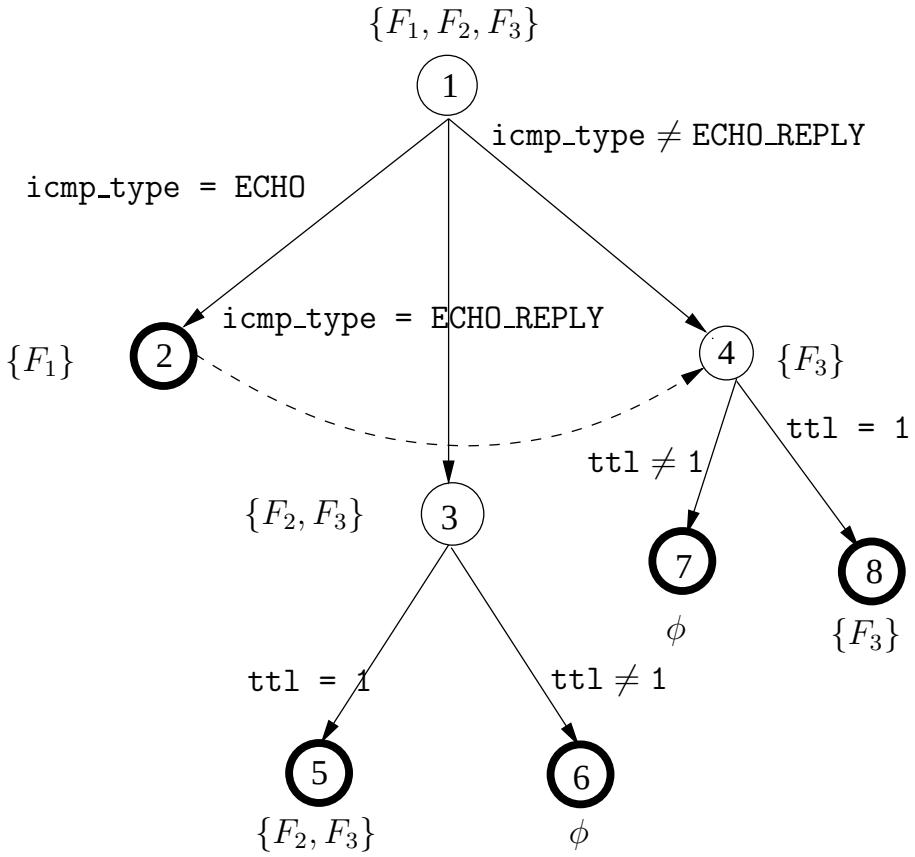

Figure 2: A nondeterministic classification automaton.

transition, is labeled with a more complex condition C as follows:

- In a nondeterministic automaton, C is the conjunction of negations of *a subset* of the tests on the rest of the transitions, e.g., the third transition from the start state in Figure 2.
- In a deterministic automaton, C is the conjunction of negations of *all* the tests on the rest of the transitions, e.g., the third transition from the start state in Figure 1. In this case, the "other" transition is mutually exclusive with the rest of the transitions, and hence is also called an "else" transition.

- The transitions from each automaton state are *simultaneously distinguishable,* i.e.,

  - apart from the "*other*"-transition, the tests on the rest of the transitions are mutually exclusive
  - it is possible to determine, using a single operation with $O(1)$ expected time complexity, which of the transitions out of a state is applicable to a given packet.

- Each final state S correctly identifies the match set corresponding to any packet satisfying all the tests along a path from the start state to S.

In a deterministic automaton, only one of the trasitions is taken at each state at runtime. For example, consider the deterministic automaton in Figure 1 and an *icmp echo* packet with a *ttl* of 1. At state 1, the leftmost transition corresponding to `icmp_type` = `ECHO` is chosen followed by the leftmost transition at state 2 to reach state 5. State 5 identifies that rules F_1 and F_3 match the given packet.

In a nondeterministic automaton, nondeterminism is simulated at runtime using backtracking: suppose that a packet satisfies a test T_i on one of the transitions out of an automaton state s, e.g., `icmp_type` = `ECHO` transition from the root of the NCA shown in Figure 2. If T_i does not appear in the "*other*"-transition, then the match will first proceed down the T_i transition, and then subsequently backtrack to s and then resume matching down the "*other*"-transition. This need for backtracking is depicted in Figure 2 using a dotted transition. Note that whether such backtracking will take place is independent of the success or failure of matches below the T_i-transition. No backtracking is required if $\neg T_i$ appears in the "*other*"-transition, which is the case for the `icmp_type` = `ECHO_REPLY` in Figure 2.

2.2.1 Computational Issues

The two main computational issues in construction of classification automata are its *size* and *matching time*. Although our experimental evaluation considers the number of automaton states as a measure of its size, for simplifying mathematical analysis, our discussion in this dissertation will use the automaton breadth as the size metric. Since the automaton is acyclic, and since tests

are never repeated, it can be shown that the total number of automaton states can, in the worst case, be at most S times its breadth, where S is the number of distinct tests across all the filters [24]. In practice, the factor is closer to average size of filters, which can be significantly smaller than S.

Typically, most filters contain a small number of tests, while the number of filters is large. As a result, path lengths in the automata are short as compared to its breadth. The matching time of an automaton is closely related to path lengths. In particular, the worst-case matching time equals the longest path length in a DCA. The average matching time is dependent on the distribution of packets observed at runtime, but it is common to use the average path length of a DCA as an estimate of average matching cost. In an NCA, note that at each state, two branches may have to be followed at runtime, and this has to be taken into account in computing the worst-case as well as average matching times.

In the next chapter, we introduce the notion of *condition factorization* that will play a central role in our automata construction algorithm.

CHAPTER 3

Condition Factorization

Condition factorization refers to the process of decomposing filters into combination of more primitive tests — a process that is intuitively similar to factorization of integers. This decomposition exposes those primitive tests that are common across different tests, and thus enables shared computation of these common primitive tests. To see this, consider the following rules involving different bit-mask tests on the same field x:

- F_1: $(x \,\&\, \texttt{0xd3} == \texttt{0x92})$
- F_2: $(x \,\&\, \texttt{0x3d} == \texttt{0x28})$
- F_3: $(x \,\&\, \texttt{0x11} == \texttt{0x11})$

Figure 3 shows the DCA constructed from these rules in a straightforward fashion. For ease of understanding, we omit transitions to the final state that corresponds to an empty match set and also labels on some of the transitions. At the start state, we test the value of $(x \,\&\, \texttt{0xd3})$ (taken from F_1). There are two transitions from this state – the left one corresponds to this value being equal to `0x92` and the right one for all other values. For the left child, we see that F_1 has matched and F_2 and F_3 can potentially match. For the right child, F_2 and F_3 can match but a match for F_1 is ruled out. Next, we pick the test $(x \,\&\, \texttt{0x3d})$ from F_2 at these children nodes. Continuing on in this way, we get the automaton as shown in Figure 3.

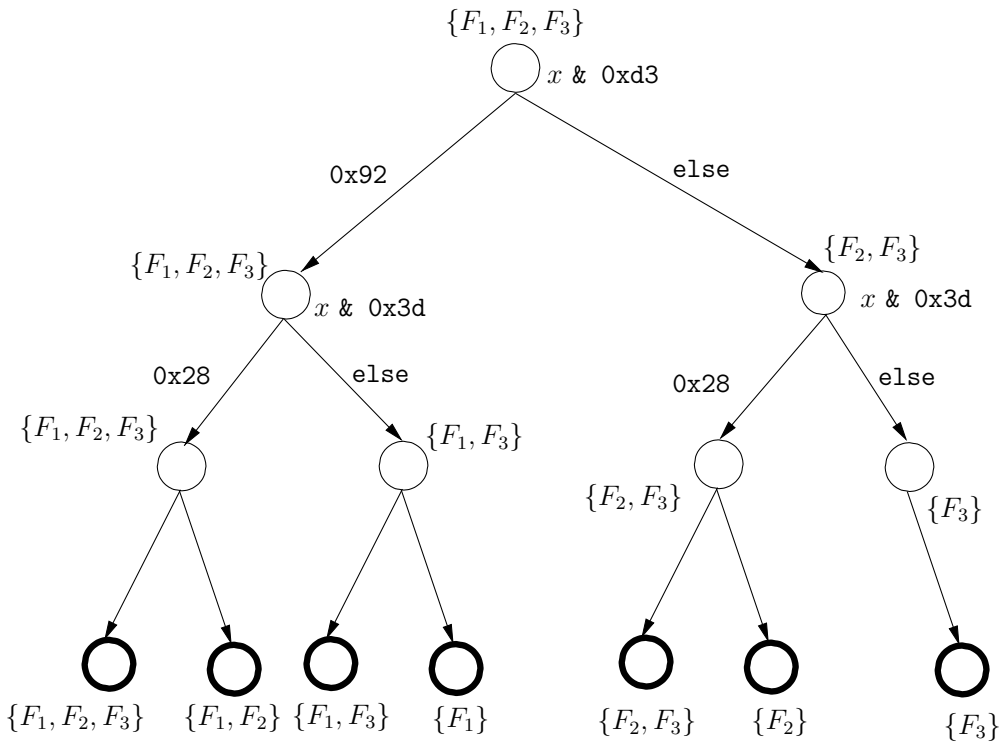

Figure 3: DCA for rules before applying condition factorization.

Using condition factorization, we can break up the tests in these rules to get the following equivalent rules.

- F_1: $(x \,\&\, \texttt{0x11} == \texttt{0x10}) \wedge (x \,\&\, \texttt{0xc2} == \texttt{0x82})$
- F_2: $(x \,\&\, \texttt{0x11} == \texttt{0x0}) \wedge (x \,\&\, \texttt{0x2c} == \texttt{0x28})$
- F_3: $(x \,\&\, \texttt{0x11} == \texttt{0x011})$

It is clear from these new rules that the test for $(x \,\&\, \texttt{0x11})$ is common in all the rules. Figure 4 shows the DCA for these rules. Here, we select the common test at the root. This results in each of the rules falling along only one transition. It is clear that the DCA obtained from the rules after applying condition factorization is more compact than the one for the original rules.

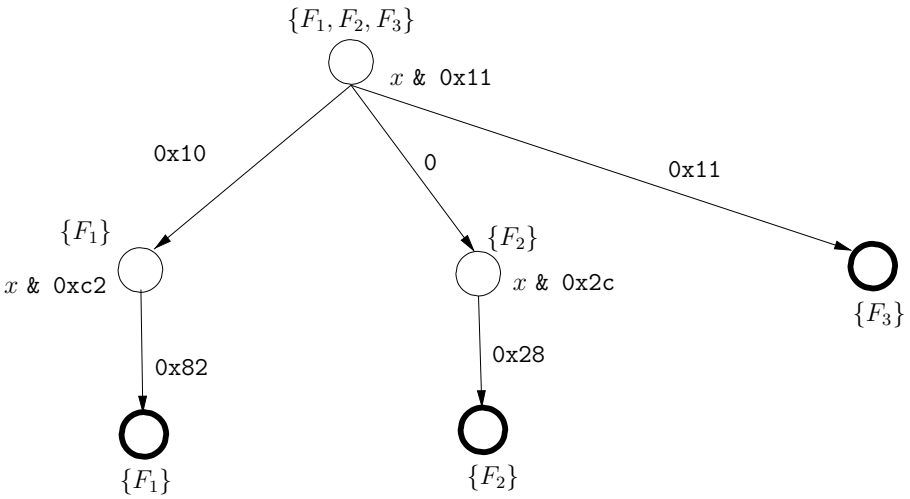

Figure 4: DCA for rules after applying condition factorization.

3.1 Residue

The basis for condition factorization is the residue operation. We first define residue and then describe how to compute it. To motivate the need for defining residues, suppose that we want to determine if there is a match for a filter C_1. Also assume that we have so far tested a condition C_2. A residue captures the additional tests that need to be performed at this point to verify C_1. We can define residue as follows:

Definition 3.1 (Residue) *For conditions C_1 and C_2, the **residue** C_1/C_2 is another condition C_3 such that $C_2 \wedge C_3 \Leftrightarrow C_1$.*

Intuitively residue operation is analogous to integer division and conjunction is analogous to product operation. The above definition follows naturally from the fact that for integers n_1, n_2, and n_3, $n_1/n_2 = n_3$ implies that $n_2 * n_3 = n_1$. Following are some examples that illustrate the residue as per this definition:

- $(x \in [1, 20])/(x \in [15, 25])$ is $(x \leq 20)$
- $(x \in [1, 20])/(x = 35)$ is $false$
- $(x \in [1, 20])/(x \neq 15)$ is $(x \in [1, 14]) \vee (x \in [16, 20])$

We can see in the last example that the result of a residue operation may have disjunctions. After we present the automata construction algorithm, *Build*, in the next chapter, it will become apparent that the presence of disjunctions, would complicate it significantly, and can adversely impact its efficiency. For instance, if we want to compute the residue of $x \in [1, 100]$ with respect to $(x \neq 3) \wedge (x \neq 6) \wedge \cdots \wedge (x \neq 99)$, then a single filter would get replaced by about 30 filters, since we do not permit disjunctions within filters. This will significantly increase the runtime of *Build* as well as the size of matching automaton. Moreover, the resulting tests (e.g., $x \in [1, 2]$) are no cheaper to test than $x \in [1, 100]$. Hence, whenever the accurate value of residue C_3 according to above definition contains disjunctions, we choose to approximate it with a stronger condition that consists of a single test. For example, we return $(x \in [1, 20])$ for $(x \in [1, 20])/(x \neq 15)$. Due to the way *Build* uses residues, this has the effect that some tests that may be implied by other previously performed tests may be repeated in the automaton. However, a syntactically identical test won't be repeated, and moreover, matches would never be announced prematurely. We redefine residue to capture this approximation as follows:

Definition 3.2 (Residue) *For conditions C_1 and C_2, the **residue** C_1/C_2 is another condition C_3 such that $C_2 \wedge C_3 \Rightarrow C_1$.*

Computing residue as per Definition 3.2 leaves the possibility that C_3 can be too strong and hence, not useful for our purpose of increasing sharing between tests. For example, this definition can always be satisfied by setting C_3 to $false$. This leads us to our final definition of residue (Definition 3.3). This is the definition that we use in the rest of the dissertation.

Definition 3.3 (Residue) *For conditions C_1 and C_2, the **residue** C_1/C_2 is another condition C_3 such that:*

(1) $C_2 \wedge C_3 \Rightarrow C_1$, and

(2) $C_1 \wedge C_2 \Rightarrow C_3$.

For a filter set, $\mathcal{F}/C = \{F/C | F \in \mathcal{F} \wedge F/C \neq false\}$.

The condition (2) in the definition ensures that C_3 cannot be too strong. Definition 3.3 also defines the residue of a filter set $\mathcal{F}$ w.r.t. a condition C. Intuitively, it means that the $\mathcal{F}/C$ is the set of the residue of each filter in $\mathcal{F}$ w.r.t C that are not $false$.

3.1.1 Computing Residue

In Figure 5 we specify the rules to compute residue (according to Definition 3.3) for the types of tests that are present in our application domain (Definition 2.1). In the figure, the notation $\overline{x}$ denotes bit-wise complement of x, while & denotes bit-wise "and" operation. In addition, inequalities are expressed using interval constraints, e.g., $x \leq 7$ is represented as $x \in [-\infty, 7]$, if x is an integer-valued variable. Note that a single interval constraint can represent a pair of inequalities involving a single variable, e.g., $(x \leq 7) \wedge (x > 3)$ can be represented as $x \in [4, 7]$.

For any pair of tests T_1 and T_2, the first row in the table that matches the structure of T_1 and T_2 yields the value of T_1/T_2. In addition, the value of T_3 in a row can be used only when the constraint in the last column is satisfied. We illustrate residue computation using several examples:

- $(x \neq a)/(x = a)$ is $false$, as given by the second row in the table (which defines $T/\neg T$).
- $(x = 5)/(x\&\texttt{0x3} \neq 1)$ is $false$, as given by the 5th row.
- for $(x = 5)/(x\&\texttt{0x3} \neq 0)$, 5th row is no longer applicable since the condition $c\&c_1 = c_2$ does not hold. (Here, $c = 5$, $c_1 = \texttt{0x3}$, and $c_2 = 0$.) Hence the applicable row is the last row, which yields $(x = 5)/(x\&\texttt{0x3} \neq 0) = (x = 5)$. The result is understandable: although the two conditions are compatible with each other, the test $x\&\texttt{0x3} \neq 0$ does not contribute to proving $x = 5$.
- $(x \in [1, 10])/(x \neq 5)$ is also given by the last row to be $(x \in [1, 10])$.

T_1	T_2	T_1/T_2	Conditions
T	T	$true$	
T	$\neg T$	$false$	
T	$x = c$	$T[x \leftarrow c]$	
$x = c$	$x \,\&\, c_1 = c_2$	$x \,\&\, \overline{c_1} = c \,\&\, \overline{c_1}$	$c \,\&\, c_1 = c_2$
		$false$	$c \,\&\, c_1 \neq c_2$
$x = c$	$x \,\&\, c_1 \neq c_2$	$false$	$c \,\&\, c_1 = c_2$
$x = c$	$x \in [c_1, c_2]$	$false$	$c \notin [c_1, c_2]$
$x \neq c$	$x \,\&\, c_1 = c_2$	$x \,\&\, \overline{c_1} \neq c \,\&\, \overline{c_1}$	$c \,\&\, c_1 = c_2$
		$true$	$c \,\&\, c_1 \neq c_2$
$x \neq c$	$x \,\&\, c_1 \neq c_2$	$true$	$c \,\&\, c_1 = c_2$
$x \neq c$	$x \in [c_1, c_2]$	$true$	$(c < c_1) \vee (c > c_2)$
$x \in [c_1, c_2]$	$x \in [c_3, c_4]$	$true$	$c_1 \leq c_3 \leq c_4 \leq c_2$
		$x \in [-\infty, c_2]$	$c_1 \leq c_3 \leq c_2 \leq c_4$
		$x \in [c_1, \infty]$	$c_3 \leq c_1 \leq c_4 \leq c_2$
		$x \in [c_1, c_2]$	$c_3 \leq c_1 \leq c_2 \leq c_4$
		$false$	$(c_2 < c_3) \vee (c_4 < c_1)$
$x \in [c_1, c_2]$	$x \,\&\, c_3 = c_4$	$false$	$c_4 > c_2$
$x \,\&\, c_1 = c_2$	$x \,\&\, c_3 = c_4$	$x \,\&\, (c_1 \,\&\, \overline{c_3}) = (c_2 \,\&\, \overline{c_3})$	$c_2 \,\&\, c_3 = c_1 \,\&\, c_4$
		$false$	otherwise
$x \,\&\, c_1 = c_2$	$x \in [c_3, c_4]$	$false$	$c_2 > c_4$
$x \,\&\, c_1 \neq c_2$	$x \,\&\, c_3 = c_4$	$x \,\&\, (c_1 \,\&\, \overline{c_3}) \neq (c_2 \,\&\, \overline{c_3})$	$c_2 \,\&\, c_3 = c_1 \,\&\, c_4$
		$true$	otherwise
$x \,\&\, c_1 \neq c_2$	$x \in [c_3, c_4]$	$true$	$c_2 > c_4$
T	T'	T	

Figure 5: Computation of Residue on Tests.

To illustrate residues on filter sets, consider

$$\mathcal{F} = \{F_1 : (x = 5),\ F_2 : (x = 7),\ F_3 : (x < 10)\}.$$

Then

- $\mathcal{F}/(x = 5) = \{F_1 : true,\ F_3 : true\}$

- $\mathcal{F}/(x < 7) = \{F_1 : (x = 5),\ F_3 : true\}$

Finally, we describe how to compute residues of complex conditions. In this regard, we note that *Build* does not need to compute residues with respect to conditions that contain disjunction operations, hence we don't consider that case below:

- $(C_1 \oplus C_2)/C_3 = (C_1/C_3) \oplus (C_2/C_3),\ \text{for} \oplus \in \{\wedge, \vee\}$
- $C_1/(C_2 \wedge C_3) = (C_1/C_2)/C_3$

Using this definition, we can compute:

- $((x > 2) \vee (y > 7))/(x = 5)$ is $true$, and
- $((x > 2) \wedge (y > 7))/(x = 5)$ is $(y > 7)$.

3.2 Condition Factorization in Automata Construction

Condition factorization plays a key role in the *Build* algorithm that is explained in the next chapter. For each state, *Build* maintains two sets: (i) *match set* that consists of all filters for which a match can be announced at that state, and (ii) *candidate set* that consists of those filters that haven't completed a match, but future matches can't be ruled out either. For a state s the candidate set is denoted by $\mathcal{C}_s$ and the match set by $\mathcal{M}_s$.

Given a state s of a matching automaton for a filter set $\mathcal{F}$, we denote the conjunction of tests on the path from the start state to s by $\mathcal{P}_s$. We can compute the match set $\mathcal{M}_s$ corresponding to an automata state s using the following steps:

- $\mathcal{M}_1 = \{M \in \mathcal{F}/\mathcal{P}_s | (M = true)\}$, i.e., $\mathcal{M}_1$ consists of those filters that are implied by the conditions examined on the automaton path reaching s.

- $\mathcal{M}_2 = \{M \in \mathcal{M}_1 | \neg \exists M' \in \mathcal{F}/\mathcal{P}_s \; Pri(M') > Pri(M)\}$, i.e., $\mathcal{M}_2$ is obtained by deleting those filters from $\mathcal{M}_1$ for which a future match with higher priority filters can't be ruled out.

- $\mathcal{M}_s$ is obtained by considering filters with equal priorities in $\mathcal{M}_2$, and deleting all but one of them.

Now, $\mathcal{C}_s$ can be computed using the following two equations:

$$\mathcal{C}_s = \mathbf{C}(\mathcal{F}/\mathcal{P}_s, \mathcal{M}_s)$$

$$\mathbf{C}(\mathcal{F}, \mathcal{M}) = \{C \in \mathcal{F} \mid \neg \exists M' \in \mathcal{M} \text{ with } Pri(M') \geq Pri(C)\}$$

These equations can be interpreted procedurally as follows. First, identify the list of all filters that are compatible with the automaton path reaching s. Next, eliminate filters that are superseded by higher (or equal) priority filters for which a match has already been completed. We maintain only the residuals of the original filters in $\mathcal{C}_s$ and $\mathcal{M}_s$, after factoring out the tests performed on the path from the root of the automaton to the state s. Hence, we are conveniently keeping track of those tests in each filter that haven't yet been performed. (Or more accurately, we are keeping track of those tests that aren't already known to be satisfied.)

To understand this, let us see how residues are used in constructing the DCA in Figure 1 from the rules in Section 2.1. At each state we need to select a field to test and decide the trasitions out of the state. The criteria for selecting this field are explained in Chapter 5. Suppose, at state 1, we decide to test `icmp_type`. Then we create transitions corresponding to the values for this field in the filter set. In our example we create transitions corresponding to (`icmp_type` = `ECHO`) and (`icmp_type` = `ECHO_REPLY`). We also create the `else` transition which corresponds to all other values for `icmp_type`. Now as described above, we compute the residue of the filter set with the test on each transition to get the match set and candidate set of the next state. For instance, at state 2, we get the match set as $\{F_1 : \texttt{true}\}$. Note that the condition component of F_1 has become `true` since we computed the residue of

the original condition (i.e., (`icmp_type` = `ECHO`)) with respect to the condition (`icmp_type` = `ECHO`) on the path from the automaton root to state 2. In addition, note that we can rule out a match for F_2 at this state, but a match for F_3 is still possible. Thus, the candidate set for this state is $\{F_3 : (\texttt{ttl} = 1)\}$. In Figures 1 and 2, we have annotated final states with match sets, and non-final states with the union of match and candidate sets. In the next chapter, we describe this automata construction algorithm in detail.

CHAPTER 4

Matching Automata Construction

Our algorithm *Build* for constructing a matching automata is shown in Figure 6. *Build* is a recursive procedure that takes an automaton state s as its first parameter, and builds the subautomaton that is rooted at s. It takes two other parameters: $\mathcal{C}_s$, the *candidate set* of the state s, and $\mathcal{M}_s$, the *match set* of s. For the start state, $\mathcal{C}_s$ consists of all filters in the input filter set, and $\mathcal{M}_s$ is empty.

A final state is characterized by the fact that there are no more filters left in $\mathcal{C}_s$. This condition is tested at line 2, and s is marked final, and is annotated to indicate $\mathcal{M}_s$ as its match set. If the condition at line 2 isn't satisfied, then the construction of automaton is continued in lines 5–16. First, a procedure *select* (to be defined later) is used at line 5 to identify a set of tests $T_1, ..., T_k$ that would be performed on the transitions from s. This procedure also indicates whether T_i is going to be a deterministic transition or not: in the former case d_i is set to $true$, while in the latter case, $d_i = false$. Section 5.2 explains the need to support nondeterministic transitions. Based on which T_i are deterministic, the condition T_o associated with the "other"-transition is computed on line 6: $\neg T_i$ is included in T_o iff T_i is to be a deterministic transition.

The actual transitions are created in the loop at line 7–16. At line 8, we

```
1.  procedure Build(s, C_s, M_s)
2.    if C_s is empty /* No more filters to match */
3.      then match[s] = M_s /* Annotate final state with match set */
4.    else
5.      (D, T) = select(C_s) /* T_i ∈ T is tested on ith transition */
            /*  d_i ∈ D indicates if this transition is deterministic */
6.      T_o = {⋀_{d_i∈D|d_i=true} ¬T_i}
            /* Compute test corresponding to the "other"-transition */
7.      for each T_i ∈ (T ∪ {T_o}) do
8.        C_i = C_s/T_i
9.        if ((T_i ≠ T_o) ∧ ¬d_i) then C_i = C_i − C/T_o endif
            /* For a nondeterministic transition, do not duplicate */
            /* filters from the "other" branch */
10.       compute M_{s_i} and C_{s_i} from C_i and M_s
11.       if a state s_i corresponding to (C_{s_i}, M_{s_i}) isn't present
12.         create a new state s_i
13.         Build(s_i, C_{s_i}, M_{s_i})
14.       endif
15.       create a transition from s to s_i on T_i
16.     end
17.   endif
```

Figure 6: Algorithm for Constructing Matching Automaton

compute the subset $\mathcal{C}_i$ of filters in $\mathcal{C}_s$ that are compatible with T_i. However, if this is going to be a nondeterministic transition, then a match would be tried down the transition labeled T_i and then subsequently down the "other"-transition. For this reason, we can eliminate from $\mathcal{C}_i$ any filter that will be considered on the "other"-transition. This elimination is performed on line 9. At line 10, $\mathcal{M}_{s_i}$ and $\mathcal{C}_{s_i}$ for the new state are computed. (The procedure for computing match and candidate sets was described in Section 3.2.)

Since the behavior of *Build* is determined entirely by the parameters $\mathcal{C}_s$ and $\mathcal{M}_s$, two invocations of *Build* with the same values of these parameters will yield identical subautomata. Hence a check is made at line 11 to examine if an automaton state already exists corresponding to $\mathcal{C}_{s_i}$ and $\mathcal{M}_{s_i}$, and if not, a new state is created at line 12, and *Build* recursively invoked on this state.

Finally, a transition to this state is created at line 15.

The algorithm presented in this chapter incorporated two main optimizations to reduce automaton size and matching time, both derived from our definition of condition factorization: detecting and sharing equivalent states, and avoiding repetition of (semantically) redundant tests. In the next chapter, we present techniques for realizing the *select* function that yields significant additional reduction in automata size.

CHAPTER 5

Techniques for Realizing *Select*

Definition of *select* amounts to determining the test that should be performed at a particular state of the automaton. Since the test identifies the packet field to be examined, *select* can be viewed as defining an order of examination of packet fields. Not all orders of examination may be acceptable, since some packet fields (e.g., the protocol field) may need to be examined before others (e.g., the port field). We will describe in section 6.1.3 how our type system captures such ordering constraints among tests. Our implementation of *select* ensures that these constraints are respected.

The simplest approach for defining *select* is to test the fields in the order of their occurrence in a network packet, as done in some of the previous works [3, 7]. We call such a traversal order as *left-to-right traversal* and refer to an automaton using this traversal order as *L-R automaton.* A better strategy, called *adaptive traversal,* was first proposed in the context of term-matching [24], and was then generalized to deal with binary data in [11]. In the terminology of this dissertation, an adaptive traversal would select a set of tests $\mathcal{T}$ at an automaton state s as follows. It identifies a packet field x that occurs in every filter in $\mathcal{C}_s$. (If no such field can be found, it falls back to another choice, e.g., choosing the left-most field that hasn't yet been examined.) Now, $\mathcal{T}$ consists of all tests on x that occur in any of the filters in $\mathcal{C}_s$.

Since adaptive traversal was developed in a context where the tests were all restricted to be simple equalities with constants, it is easy to see that the set

$\mathcal{T}$ described above consists of tests that can be simultaneously distinguished[1], and hence can form the transitions from s. Moreover, it has been shown [24] that, as compared to other choices, this choice of transitions will simultaneously reduce the automaton size as well as matching time. Unfortunately, none of these hold in the more general setting of packet classification, where disequalities and inequalities also need to be handled. For instance, consider a candidate set that consists of two filters $(x \neq 25)$ and $(x < 1024)$. These tests are not simultaneously distinguishable. Moreover, neither of these tests contributes towards verifying a match with the other. More generally, it can be shown that, in the presence of disequality and inequality tests, the choices that decrease automaton size do not necessarily decrease matching time (and vice-versa). We therefore focus first on a criterion for reducing automaton size.

5.1 Partitioning Tests

The main reason for the blow-up in size of automata is the duplication of rules. Consider a node that examines a field that is not present all the rules. If the node has two children, rules that do not examine this field would need to be duplicated across these children. Repeated duplication leads to automata whose size, in the worst-case, is exponential in the number of rules [24]. Our first strategy aims to avoid the blow-up in size by picking tests that do not lead to duplication in the children nodes. We formalize this notion using the the following definition:

Definition 5.1 (Partitioning Set) *A set $\mathcal{T}$ of conditions is said to be a **partitioning set** for a filter set $\mathcal{F}$ iff for every $F \in \mathcal{F}$ there exists at most one $T \in \mathcal{T}$ such that F belongs to the candidate set of $\mathcal{F}/T$.*

The set $\mathcal{T} = \{x = 5, x = 6, (x \neq 5) \wedge (x \neq 6)\}$ is partitioning for the filter set $\mathcal{C} = \{x = 5, x = 6, x > 7\}$, but not for $\{x = 6, x > 4\}$. This means if we create 3 outgoing transitions corresponding to the three tests in $\mathcal{T}$ from

[1]Recall that simultaneous distinguishability refers to the ability to identify the matching transition in $O(1)$ expected time.

an automata state s with the candidate set $\mathcal{C}$, none of the filters in $\mathcal{C}$ will be duplicated among the children of s. As a result, in an automaton that uses only partitioning tests, the candidate sets (as well as the match sets) associated with the leaves will be disjoint. Since there are at most n disjoint subsets of a set of size n, it immediately follows that any automaton that is based entirely on partitioning tests will have at most $O(n)$ breadth.

5.2 Ensuring Polynomial-Size Automata

Since partitioning tests may not always exist, it may be necessary to choose non-partitioning tests. This choice introduces overlaps among the candidate sets of sibling states in the automaton. These overlaps, in turn, mean that at any level in the automaton, there may be as many as 2^n distinct candidate sets. Thus, the breadth of the automaton can become exponential in the number of filters. Exponential *lower bounds* have previously been established even in the simple case where all tests are restricted to be equalities [24]. Although some of the previously developed techniques can avoid such explosion, this has been accomplished at the cost of introducing significant backtracking at runtime [18, 7, 3, 4], especially for the kinds of filters that occur in the context of intrusion detection. Other techniques avoid exponential size by introducing $O(n)$ operations for each transition at runtime, as they require runtime maintenance of match sets [21, 11]. With large filter sets that are often found in enterprise firewalls and network intrusion detection systems, $O(n)$ time complexity for transitions becomes unacceptable.

We present a new technique that can provide a polynomial size bound, while limiting nondeterminism in practice. Indeed, any desired polynomial bound $P(n)$ can be achieved by our technique. However, by using a larger bound, e.g., n^2 instead of $n \log n$, one can obtain deterministic automata in almost all cases.

Our technique is based on the observation that the breadth of subautomaton rooted at s can be captured, in terms of the sizes of candidates sets

associated with s and its children, using the recurrence

$$B(|\mathcal{C}_s|) = \sum_{i=1}^{k} B(|\mathcal{C}_{s_i}|),$$

where $B(1) = 1$. Let $P(n)$ be the desired polynomial on n that bounds the automaton size. Based on the above recurrence, we can show, by induction on the height of s that the bound will be satisfied as long as the following condition holds at every state s of the automaton.

$$P(|\mathcal{C}_s|) \geq \sum_{i=1}^{k} P(|\mathcal{C}_{s_i}|) \tag{1}$$

By selecting tests that satisfy this constraint, our implementation of *select* ensures that the automaton size will be $O(P(n))$. If no such test can be found, our technique picks a test that comes the closest to satisfying this constraint, and then makes some of the outgoing transitions nondeterministic so as to ensure that sizes of candidate sets associated with the descendant automaton states satisfy the above constraint. Recall from line 9 of *Build* that making a test $\mathcal{T}_i$ nondeterministic enables us to avoid overlaps between $\mathcal{C}_i$ and $\mathcal{C}_o$. So, our algorithm makes one or more transitions out of an automaton state nondeterministic until Inequality 1 is satisfied. In our implementation, we have set $P(n)$ to be n^2, which guarantees a quadratic worst-case automaton size.

The above technique can be extended further: rather than looking at one level of the automaton at a time, we could examine all of the ancestors of a state s, and ensure that collectively, they stay within the budget permitted by $P(n)$. This would permit a greater degree of overlap at s if the degree of overlap among (the children of) the ancestors of s was smaller than the budget.

To understand the importance of the above technique, note that a purely deterministic technique ensures good performance at runtime, but risks catastrophic failure on large rule sets that cause an exponential blow up — memory

will be exhausted in that case and hence the rule set can't be supported. In contrast, our approach converts this catastrophic risk into the less serious risk of performance degradation. Unlike previous techniques for space reduction that led to increases in runtime in practice, performance degradation remains a theoretical possibility with our technique, rather than something observed in our experiments. (This is because of the fact that with the rule sets we have studied in our experiments, the quadratic bound was never exceeded, and hence nondeterminism was not introduced.)

5.3 Benign Nondeterminism

For our final space-reduction technique, we define the concept of benign nondeterminism, which enables us to benefit from the space-savings enabled by nondeterminism *without incurring any performance penalties.* It is based on the following notion of *independence* among filter sets.

Definition 5.2 (Independent Filters) *Two filters F_1 and F_2 are said to be* ***independent*** *of each other if*

- *$Pri(F_1)$ and $Pri(F_2)$ are either equal or incomparable,*
- *for every test T in F_1, $F_2/T = F_2$, and*
- *for every test T in F_2, $F_1/T = F_1$.*

$\mathcal{F}_1$ and $\mathcal{F}_2$ are said to be independent if $\forall F_1 \in \mathcal{F}_1, \forall F_2 \in \mathcal{F}_2$, F_1 and F_2 are independent.

Suppose that there is a filter set $\mathcal{F}$ that can be partitioned into two independent subsets $\mathcal{F}_1$ and $\mathcal{F}_2$. We can then build separate automata for $\mathcal{F}_1$ and $\mathcal{F}_2$. Packets can now be matched using the first automaton and then the second one. For the packet-filtering case, characterized by equal priorities among all filters, we need to match with the second automaton only if the first automaton reports no matches. From the above definition, it is clear that the tests

appearing in the two automata are completely disjoint, and hence no decrease in runtime can be achieved by constructing a single automaton for $\mathcal{F}$.

Our experiments show that the above technique leads to dramatic reductions in space usage. The intuition for this is as follows. If F_1 and F_2 are independent, then a packet may match F_1, F_2, both, or neither. A deterministic automaton must have a distinct leaf corresponding to each of these possibilities. Extending this reasoning to independent filter sets, if an automaton for the set $\mathcal{F}_1$ has k_1 states, and the automaton for $\mathcal{F}_2$ has k_2 states, then a deterministic automaton for $\mathcal{F}_1 \cup \mathcal{F}_2$ will have $k_1 * k_2$ states. In contrast, using benign nondeterminism, the size is limited to $k_1 + k_2$. If there are m independent filter sets, then the use of benign nondeterminism can reduce the automaton size from a product of m numbers to their sum.

The second reason for significant reductions in practice, especially in the case of network intrusion detection system rules, is as follows. After examining some of the fields that are common across many rules, as we get closer to the automaton leaf, independent sets arise frequently. For instance, we may be left with one set that examines only the destination port, another set that examines only the source port, yet another set that examines only the destination network, and so on. Thus, independent rule sets tend to arise frequently, and lead to massive increases in space usage if they are not recognized and exploited using our benign nondeterminism technique.

There is a simple algorithm for checking if $\mathcal{F}$ contains two independent subsets. First, partition $\mathcal{F}$ into subsets such that any two rules F_1, F_2 such that $Pri(F_1) > Pri(F_2)$ are in a single subset. Now, these subsets are taken two at a time, and merged if they are *not* independent. This process is repeated until no more merges are possible. If there are multiple subsets left at this point, then these subsets are independent.

To deal with benign nondeterminism, the interface between *select* and *Build* needs to be extended so that the former can return a set of independent filter sets $\{\mathcal{F}_1, \ldots, \mathcal{F}_k\}$, instead of a test set. At this point, *Build* will create a k-way nondeterministic branch. On the ith branch, it will invoke $Build(s_i, \mathcal{F}_i, \mathcal{F}_i \cap \mathcal{M}_s)$.

5.4 Improving Matching Time

To reason about matching time, we need to define a function that assigns computational costs to each test. A simple cost model is one that assigns unit cost to all tests. Note that such a measure would treat tests on 1-bit fields the same as on 32- or 64-bit fields. While this may seem reasonable, it does not capture the intuition that checking a test $y\&\texttt{0xff} = 3$ contributes partially towards checking $y = \texttt{0x703}$. For this reason, we prefer to use a measure that assigns a cost of r to tests involving r-bit quantities. In this case, $cost(y\&\texttt{0xff} = 3)$ will be 8, while $cost(y = \texttt{0x703})$ will be 16, assuming y is a 16-bit field. However, to simplify our presentation, we will use the uniform cost model below, and ignore priorities. Our technique for reducing matching time is based on the following notion:

Definition 5.3 (Utility) *The utility $U_s(T, F)$ of a test T at an automaton state s for a filter $F \in \mathcal{C}_s$ is*

- *0, if a match for F is ruled out when T is satisfied*
- *$cost(F) - cost(F/T) - cost(T)$, otherwise.*

The utility U_s of the set $\mathcal{T}$ of tests on the transitions from s is the weighted average,

$$\frac{\sum_{F \in \mathcal{C}_s} \sum_{T \in \mathcal{T}} U_s(T, F)}{|\mathcal{T}| * |\mathcal{C}_s|}.$$

We assume that filters do not contain redundant tests. In this case, the utility value can never be greater than zero. A negative value, which indicates that potentially unnecessary computation was carried out, is characterized by the fact that a test T costs more to perform than the cost it takes away from future tests that need to be performed for verifying a match for F. The lowest possible value of $U(T, F)$ is $-cost(T)$.

Our technique for improving matching time is based on choosing tests that have high utilities. Our implementation of *select* places more importance on size reduction than matching time. As such, it chooses test sets that maximize utility among those that minimize size.

CHAPTER 6

Implementation: Putting It All Together

We use a high-level language to specify packet classifiers. We have developed a compiler that uses the techniques presented in Chapters 4 and 5 to construct packet classification automaton from the given packet classifier specification. Once the automaton is constructed, our compiler generates C-code corresponding to the automaton, which is then compiled into native code using a C-compiler. This chapter describes the high-level specification language and how the techniques presented previously are implemented in the back-end of the compiler. In the last part of this chapter, we describe how to integrate the code generated by the compiler into existing security applications.

6.1 Language for Specifying Packet Classifiers

The performance of packet classification can be improved by using native code instead of interpreted code. Native code can be generated from packet classification code written in a low-level language like C. The most straight forward way to write such code is by treating the packet as a sequence of bytes. There are many problems with this approach. To access any field of the packet, the offset of that field from the start of the byte sequence has to be calculated. This way of accessing fields with offset calculations has many potential pitfalls.

For example, to access the source port field of tcp header, one needs to first ensure that the packet is a tcp packet. The offset for tcp source port depends on the length of the variable-length options field of ip header. Also, the bytes at those offsets need typecasting to `unsigned short` and conversion to the host order. It is clear that writing such code is very clumsy and error-prone.

A better approach is to overlay the packet header structure on the byte sequence and then access packet header fields as fields of the structure. Even this approach does not solve the problem completely due to the presence of variable length fields and the need to perform protocol decoding before accessing any field. Another approach is to use a special language developed explicitly for packet processing. Such a language can have a hand-crafted type checker for particular network protocols or have a generic type checker that supports different network protocols. In the former approach, the packet structure for supported protocols are hard coded into the compiler. This approach is very rigid and supporting new protocols requires modification to the compiler. We use the latter approach which is more flexible and extensible.

Our specification language is based on type systems that have been developed for handling network packets [5, 23]. These type systems can capture packet structures while providing the capabilities to dynamically identify packet types at runtime. This enables an approach where we can generate native code from the packet classifier specification, and be assured that this code can be safely loaded and run within the kernel if needed. Our techniques for generating space- and time-efficient packet classification automata have been implemented in the back-end of the compiler for this language.

A specification in our language consists of type declarations for describing packet structures and packet classification rules. In the following sections we describe each of these components of specifications.

6.1.1 Packet Type Description

We specify the type of packet headers using declarations that are similar to `struct` declarations in C-language. Below is the declaration of Ethernet

header:

```
#define ETH_LEN 6
struct ether_hdr {
  byte           e_dst[ETH_LEN];       /* Ethernet destination */
  byte           e_src[ETH_LEN];       /* Ethernet source */
  unsigned short e_type;               /* protocol of carried data */
};
```

The nested structure of protocol header can be captured using a notion of inheritance. For example, an IP header can be considered as a sub-type of Ethernet header with extra fields to store information specific to IP protocol. The specification language permits multilevel inheritance to capture protocol layering. Inheritance is augmented with constraints to capture conditions where the lower layer protocol data unit (PDU) has a field identifying the higher layer data that is carried over the lower layer protocol. For instance, IP header derives from Ethernet header only when `e_type` field in the Ethernet header equals `0800h`.

```
#define ETHER_IP 0x0800
struct ip_hdr : ether_hdr with e_type == ETHER_IP {
  bit            version[4];           /* IP version */
  bit            ihl[4];               /* header length */
  byte           tos;                  /* type of service */
  unsigned short tot_len;              /* total length */
  unsigned short id;                   /* identification */
  unsigned short flags_and_frag;       /* flags and fragment offset */
  byte           ttl;                  /* time to live */
  byte           protocol;             /* protocol */
  short          check_sum;            /* header checksum */
  unsigned int   s_addr;               /* source IP address */
  unsigned int   d_addr;               /* destination IP address */
};
```

Similarly, the following defines TCP over IP:

```
#define TCP 0x10
struct tcp_hdr: ip_hdr with protocol == TCP {
    unsigned short  tcp_sport;         /* source port number */
    unsigned short  tcp_dport;         /* destination port number */
    unsigned int    tcp_seq;           /* sequence number */
    unsigned int    tcp_ackseq;        /* acknowledgement number */
    bit             tcp_hlen[4];       /* header length */
    bit             tcp_reserved[4];   /* reserved */
    byte            tcp_flag;          /* tcp flags */
    unsigned short  tcp_win;           /* window size */
    unsigned short  tcp_csum;          /* checksum for header & data */
    unsigned short  tcp_urp;           /* urgent pointer */
};
```

To capture the fact the same higher layer data may be carried in different lower layer protocols, the language provides a notion of disjunctive inheritance. The semantics of the disjunctive inheritance is that the derived class inherits fields from exactly one of the possibly many base classes. The following

```
struct ip_hdr : (ether_hdr with e_type == ETHER_IP) or
                    (tr_hdr with tr_type == TOKRING_IP) {
 ...
}
```

represents the fact that IP may be carried within an Ethernet or a token ring packet. Now we can define a variable of type ether_hdr as follows:

```
ether_hdr p;
```

The fields of the headers are accessed similar to fields of a structure. For example, `p.s_addr` refers to the IP source address and `p.tcp_sport` refers to the TCP source port. We explain in Section 6.1.3 how the compiler ensures type safety of this access at runtime. From now on, we drop the variable part and just refer to the field names for ease of understanding. For example, we use `s_addr` instead of `p.s_addr`.

6.1.2 Rules

The rules are of the form *cond* $\longrightarrow$ *act*, where *act* specifies the action to be taken on a packet that matches the condition *cond*. The condition is a conjunction of tests on packet fields. The language supports various tests like equality, disequality, and inequality along with bit-masking operations on packet fields as described in Section 2.1. A packet matches a rule if all tests in the rule succeed. If multiple rules match at the same time, actions associated with each rule are launched. Labels are used to identify the rules. Consider the following rules:

```
R1 : (p.s_addr & 0xffffff00 == 0xc0a80200) && (p.d_addr == 0xc0a80100)
    && (p.tcp_dport == 80) ⟶ alert(R1);
R2 : (p.s_addr & 0xffffff00 == 0xc0a80200) && (p.d_addr == 0xc0a80100)
    && (p.ttl >= 220) ⟶ alert(R2);
```

The first test in the rule `R1` is equivalent to checking whether the source address of the packet belongs to `192.168.2.0/24`. Here, `0xc0a80200` is the hex representation of `192.168.2.0` and `0xffffff00` corresponds to the 24-bit subnet mask. This rule further checks that destination address is `0xc0a80100` (`192.168.1.0`) and destination port is `80`. The second rule `R2` also contains the same tests on source and destination address. In addition, it tests if the time-to-live (`ttl`) value is greater than `220`.

Priority relation over the rules can be specified using `priority` declarations. The priority relation can be a partial order. Figure 7 shows an example of how the priority relation is defined for some rules $\{\texttt{R1}, \texttt{R2}, ..., \texttt{R6}\}$. `R1` has a higher priority than `R2`. The priorities for $\{\texttt{R1}, \texttt{R2}\}$ cannot be compared with any of $\{\texttt{R3}, \texttt{R4}, \texttt{R5}, \texttt{R6}\}$. `R3` has higher priority than $\{\texttt{R4}, \texttt{R5}\}$. `R4` and `R5` have equal priority that is higher than `R6`. These priorities can be specified in our language as follows:

```
priority {R1, R2};
priority {R3, {R4, R5}, R6};
```

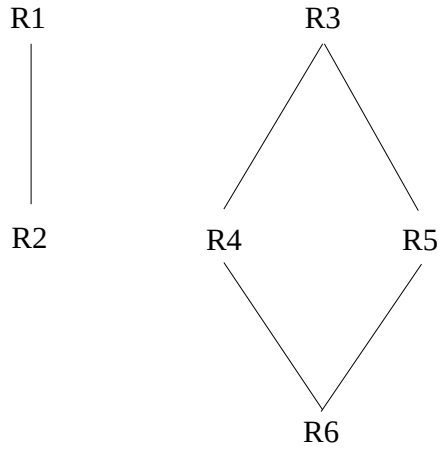

Figure 7: Priority relation between rules.

6.1.3 Compilation

Rule compilation involves the following steps:

- *Introduction of constraints based on type declarations:* Based on type declarations of network packets, our rule compiler *automatically* inserts the constraints associated with a structure before any member of that structure can be accessed. These constraints are identified as *preconditions* that must be satisfied before a certain test can be performed. A precondition P associated with a test T is denoted as $\langle P \rangle : T$. For instance, the test

  $$(\texttt{s\_addr} \ \& \ \texttt{0xff000000} == \texttt{192})$$

  would be converted into:

  $$\langle \texttt{e\_type} == \texttt{ETHER\_IP} \rangle : (\texttt{s\_addr} \ \& \ \texttt{0xff000000} == \texttt{192})$$

  This ensures that the `s_addr` is tested only after `e_type` = `ETHER_IP` has been verified. Except for this constraint on ordering of tests, preconditions are handled just like other tests in our technique.

- *Construction of matching automata* from the classification rules is the most important step of our technique. Our implementation compiles the given packet classifiers into an automaton using the *Build* algorithm presented in Chapter 4. Residues are computed as specified in Section 3.1.1. Our *select* implementation proceeds as follows:

  - *select* first attempts to find a partitioning test set (Section 5.1). If several of them exist, our technique selects a set that maximizes utility (Section 5.4).
  - if no partitioning test sets exist, it examines opportunities for benign nondeterminism (Section 5.3).
  - if neither of the above steps succeed, it returns a set of tests that achieves the polynomial size target specified, as described in Section 5.2.

  In order to speed up *select*, our implementation starts by examining fields that occur in all filters in a candidate set, giving preference to those fields that contain primarily equality tests. Such fields have a high likelihood of yielding partitioning tests with zero (i.e., maximum possible) utility, at which point *select* returns this set. As mentioned earlier, any constraints regarding the order of examination of fields are enforced by *select*.

- *Generating native code from the matching automaton:* Once the automaton is constructed, our compiler generates C-code corresponding to the automaton, which is then compiled into native code using a C-compiler. The code generation is done in straight-forward manner using an if-then-else, a binary search, or a hash-based branching to implement transitions. Further, it involves mapping of field name accesses into accesses on network packets. Accesses using variable names are translated into accesses involving offsets within packets. In addition, appropriate checks on the length of the packet are added. The compiler also takes care of converting packet fields from network to host order when needed.

6.2 Integration with Snort

The packet classification algorithm that we developed can be used as plug-in replacement in many existing security tools. This allows us to use existing rule sets for these systems. This is very beneficial for tools such as Snort [22], a popular open source network intrusion detection system, that have a large established base. Snort [22] comes with default rules that are comprehensive and up-to-date. Moreover, the performance can be improved without requiring modifications to the other components of the system such as alert processing.

In this section we describe how the packet classification code generated by our technique can be used in tools like Snort.

6.2.1 Snort Language

Snort uses a simple rule-based language. Snort rules are written in a configuration file which is read when Snort starts up. A Snort rule file consists of variable declarations and rules. The variable declarations are similar to typedefs in C; the value of the variable is substituted in the rules for matching. The rules themselves consists of a *rule header* and a *rule body*.

Rule header consists of *action*, *protocol*, *ip addresses*, *ports*, and *direction operator*. Rule actions specify the action like logging or alerting that Snort should perform when a rule matches a packet. Each rule is applicable to packets belonging to a particular protocol like TCP, UDP, ICMP, or IP. For TCP and UDP rules, the header specifies the source and destination ip addresses and port fields for which the rule is to be applied. Specifying "`any`" for one of these fields means that the field in the rule matches for any value in a packet. The fields to the left of the direction operator ($\rightarrow$) are the source fields, while the ones on the right hand side are for the destination. An alternative operator, called bidirectional operator (<>), indicates that the rule is to be applied to both directions of the flow. A rule that generates an alert when it sees a `tcp` packet from `any` port on an internal host to host `192.168.1.1` on port `80` can be writtern as follows:.

```
var internal_host 192.168.2.0/24
alert tcp $internal_host any -> 192.168.1.1 80
```

where "`internal_host`" is a variable whose value is the host address `192.168.2.0`

with subnet mask of 24 bits. So any host with this subnet address matches internal_host variable.

Rule body consists of rule options which belong to one of the following categories:

(i) *meta-data* options provide information about the rule but are not used in rule matching operation,

(ii) *payload* options are concerned with tests for deep packet inspection,

(iii) *non-payload* options specify other tests including tests on packet header fields, and

(iv) *post-detection* options specify some triggers which are fired when a rule matches a packet.

Consider the rule body appended to the previous Snort rule:

```
var internal_host 192.168.2.0/24
alert tcp $internal_host any -> 192.168.1.1 80 (msg:‘‘web-attack’’;
          ttl: 5; content: ‘‘abc’’; logto: ‘‘logfile’’;)
```

In this rule, `msg` is a meta-data option that specifies the message to be generated when a packet matches this rule. `logto` is a post-detection option that specifies the file to be used for logging. `content` is a payload option which means that the rule is matched by a packet only if it contains the string "`abc`" somewhere in the payload. Further, the packet has to satisfy the constraint that `ttl` field value is equal to `5` for the rule to match. A packet can match a rule only if it matches the rule header and all the payload and the non-payload options in rule body. These options can be viewed as tests for packet classification and hence, a rule is a logical conjunction of these tests. We ignore meta-data and post-detection options in the rest of this dissertation as they are not used for packet classification.

6.2.2 Packet Classification in Snort

In this part, we consider only the rule header and the non-payload options. We defer the discussion about the payload options to Part II. Snort uses certain fields –

`source` and `destination ports` for TCP and UDP, `type` for ICMP, and `protocol` for IP rules – to divide the rule set into groups.

At runtime, Snort captures packets using *pcap* library. It uses simple packet classification to identify the rule group that a packet belongs to. All the rules in the rule group are matched in sequence against the packet. The alerting sub-system is called for the rules that match the packet.

To use our packet classification algorithm, we convert the Snort rule set to a specification for our language. We implemented a Perl based translator for converting Snort rules into a specification for our language. The translator generates the packet structure specification and generates a rule in the specification for each rule in a Snort rule file. So there is a one-to-one correspondence between the rules in the Snort rules file and the rules in our specification file. The rules in our specification contain only the tests on packet header fields. The other tests in the rules are checked by Snort itself.

For each non-payload detection option of Snort rules we generate the corresponding packet field test in our language. For example, consider the following rule in Snort,

```
alert tcp $EXTERNAL_NET any -> $INTERNAL_NET 80 (..., ttl: 5; ...)
```

This rule generates alerts for tcp packets with `ttl` field of `5` from external network to internal network on port for http (`80`). The corresponding rule in our specification language is

```
R1: (p.proto == tcp) && (p.s_addr == $EXTERNAL_NET)
     && (p.d_addr == $INTERNAL_NET) && (p.tcp_dport == 80)
     && (p.ttl == 5) ->  alert(R1)
```

We use our compiler to compile the Snort rules in our specification format into C code. The compiler generates the packet classification automaton for the rules using our techniques from Chapter 4 and 5. Then it generates the C-code for matching this automaton in a straight-forward way using *if-then-else* branching, *switch* statements, and hashing as appropriate. We use C compiler like *gcc* to generate native packet classification code in the form of a shared library. So to update the rules, all that one needs to do is to compile the rules offline and then reload the

shared library. We note that this approach is no more disruptive than that of Snort where the rules need to be re-read and recompiled.

We load the shared library containing the packet classification code when Snort starts up. At runtime, when a packet is delivered to Snort by *pcap library*, we pass on the packet to the shared library. The shared library matches the packet against all the rules and returns the rules that match. At this point control is transferred to the default Snort processing engine. From this point on, the usual Snort processing (like logging) is performed on the packet.

We note that using this approach does not modify the behavior of Snort. In particular, for any packet the modified Snort matches the same rules as the original Snort. This is because we are just changing the way packet classification is performed without changing the actual tests in a rule.

CHAPTER 7

Evaluation of Packet Classification

We evaluated the effectiveness of our techniques in the context of network intrusion detection systems (Section 7.1) and firewalls (Section 7.1.4). Our experiments were performed on a system with 1.70Ghz Pentium 4 processor and 520MB memory, running CentOS-4.2 (Linux kernel 2.6.9).

7.1 Experiments using IDS

For our experiments we use Snort [22] as it has publically available default rules that are comprehensive and up-to-date. Snort rules consist of two main components: tests involving packet fields, and content-matching operations on the payload. According to [8], packet classification and content-matching are the most expensive parts of Snort, accounting for 21% and 31% of the execution time. Within network intrusion detection systems research community, researchers have been investigating techniques for parallelizing packet-field matches as well as content matches. In this part, our evaluation focusses only on the packet field matching component of network intrusion detection systems.

Although earlier versions of Snort relied largely on sequential matching (i.e., matching a packet against one rule at a time), newer versions of Snort (specifically, version 2.0 and later) attempt to match the rules in parallel. Newer versions of Snort rely on an ad-hoc approach for parallelizing rule matches where a small set

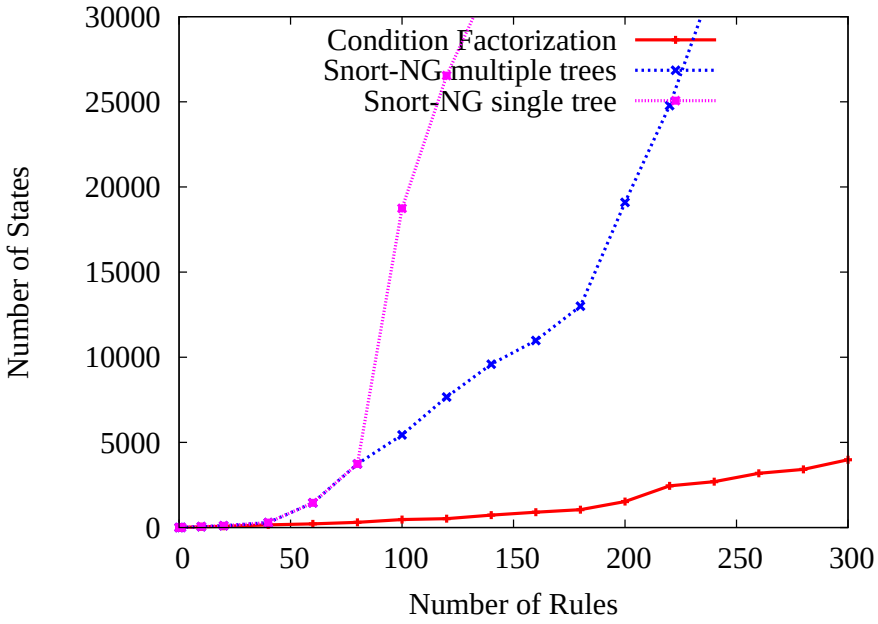

Figure 8: Automaton Size for Snort Rules

of hand-picked packet fields, such as destination ports, are tested first, but there is no systematic technique for matching other packet fields in parallel. In contrast, Kruegel and Toth developed the `Snort NG` [14] system, which demonstrated the performance gains achievable by parallelizing the rule matching. They use an entropy-based algorithm to decide which packet field to test at each node. Their technique is the only one that we are aware of that uses a sophisticated packet-classification algorithm for Snort-type rules. Hence we compare our performance results with them. To simplify this comparison, we used the default rule set that comes with `Snort NG`, which consists of 1635 rules. Since our focus is only on matching packet fields, we combined the rules that differ only in terms of payload contents. This resulted in a rule set with 305 unique rules.

7.1.1 Automaton Size

We provided `Snort NG` and our technique with the same set of 305 rules. The `Snort NG` algorithm suffers from a space explosion if all of the rules are put into a single decision tree. To cope with this, they arbitrarily divide the rules into several

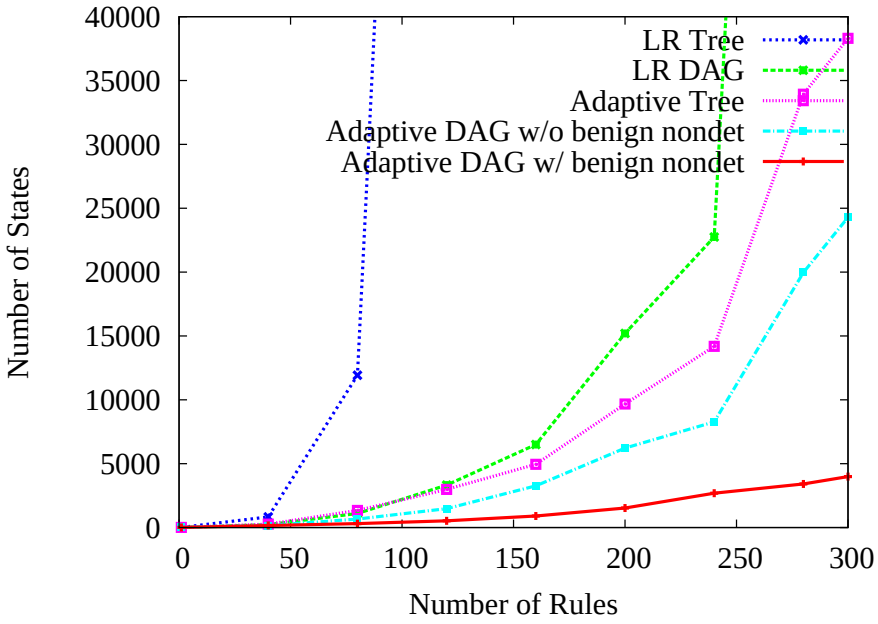

Figure 9: Effect of Optimizations on Automaton Size for Snort Rules

subsets, and build multiple decision trees, which can degrade runtime performance. Our technique builds a single deterministic automaton.

Figure 8 shows the effect of increasing the number of rules on the number of automaton states. We note that `Snort NG` decision tree has a complicated structure, where some of the states do not perform any tests, but are used to identify the type of field being tested. For our experiments we count only the states which actually perform some test. We can see from the graph that as the number of rules increases, the number of states in `Snort NG` (even after splitting the rules into multiple trees) increases much faster than our technique. For 300 rules, `Snort NG` automaton contains over 45,000 states whereas the automaton constructed by our technique has only about 4000 states. *This translates to a size reduction by a factor of about 10.*

7.1.1.1 Effect of Optimizations on Automaton Size

Figure 9 illustrates the effects of different optimizations on the automaton size. We studied different combinations of techniques: with and without sharing of equivalent states in the automata, and with different traversal orders.

- *Order of testing fields.* As compared to L-R order for examining packet fields, our technique (which uses the *select* function as described in Section 6.1.3 produces tree automata that are much smaller: for 120 rules, the L-R automaton had 150,000 states, whereas the tree automaton had less than 3000 states.

- *DAG Vs tree automata.* Our results show that DAG automata were smaller than tree automata by about 25% for our technique. Larger space reductions were achieved with DAG optimization for L-R automata, but still, L-R automata remain significantly larger than the one constructed by our technique.

- *Benign nondeterminism.* By exploiting benign nondeterminism, we were able to achieve dramatic reductions in space usage. This is because Snort contains many rules which test some common fields. Our technique prefers these common fields for testing, since they are the ones that are likely to be partitioning. Once these common fields are tested, the residual rule sets contain many independent subsets.

We point out that a combination of our techniques was necessary to achieve the size reductions we have reported. In particular, benign nondeterminism leads to large improvements in size when combined with partitioning tests. It is much less effective when used with L-R technique, since the factors contributing to the occurrence of independent filter sets do not apply in the case of L-R technique.

7.1.2 Matching Time

For measuring runtime performance, we used two sets of data. The first one consists of all packets captured at the external firewall of our laboratory that hosts about 30 hosts. Since the firewall is fully open to the Internet (i.e., the traffic is not prescreened by another layer of firewalls in the university or elsewhere), the traffic is a reasonable representative of what one might expect a network intrusion detection systems to be exposed to. Our packet trace consisted of about 21 million packets collected over a few days. Figure 10 shows the matching time taken by Snort, `Snort NG` and our technique for classifying these packets as the number of rules change.

We also used a second packet trace for performance measurement. This data corresponds to 10 days of packets from the MIT Lincoln Labs IDS evaluation data

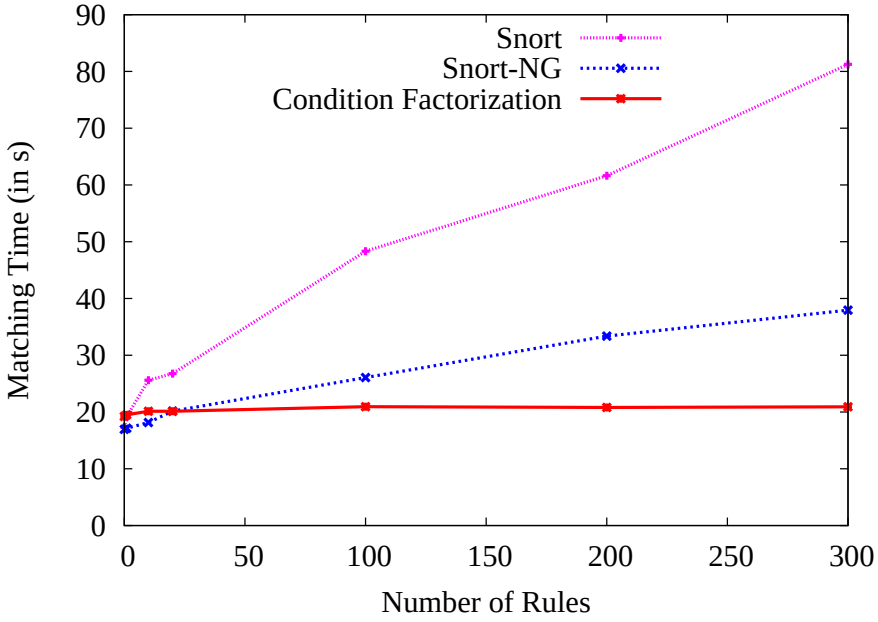

Figure 10: Matching Time for Our Lab Traffic

set [16], consisting of 17 million packets. While there has been some criticism of this data for the purpose of evaluating IDS, they primarily concern artifacts in the data that may make it easier to detect attacks. Since our focus is not on evaluating the quality of the rule set, these concerns are not that significant in our context. Moreover, we note that the results obtained with both data sets are similar.

In the Figures 10 and 11, it can be seen the matching time remains essentially constant with our technique, even as the number of rules are increased from about 10 to 300. In contrast, the matching times for Snort and `Snort NG` increase significantly with the number of rules. The base matching time for all the techniques (with no rules enabled) is basically the same, as it corresponds to the time spent by Snort to read the packets from a file and do all related processing except matching.

One of the reasons for a drastic difference in the rate of increase in matching time is due to the fact that we compile our automata into native code, whereas Snort-NG and Snort use an interpreted approach. As explained in section 6.2.2, the packet classification code that we generate is compiled into a shared library. So to update the rules, all that one needs to do is to compile the rules offline and then reload the shared library. We note that this approach is no more disruptive than

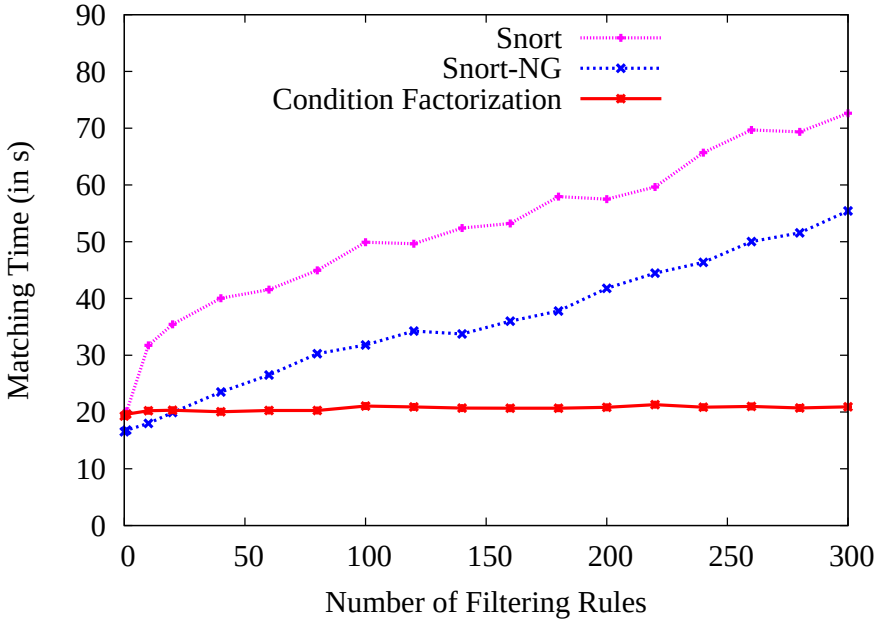

Figure 11: Matching Time for Lincoln Lab Traffic

that of Snort where the rules need to be re-read and recompiled.

7.1.3 Measuring Match Time

We now proceed to develop an implementation-independent metric for quantifying the overall matching cost of an automaton. Such a metric is preferable to raw runtimes that are heavily influenced by low-level implementation decisions. For instance, since our matching automaton is compiled into native code, it is many times faster than some of the techniques that we compare against. Thus the raw numbers don't necessarily reflect the benefits obtained using the algorithms developed in this approach, which are applicable to compiled as well as interpretation-based implementations.

Our metric is based on lower bounds on *match verification cost.* In particular, suppose that there exists a *nondeterministic matching algorithm* that can "guess" the subset of rules that match a given packet p, and then proceeds to verify the correctness of this guess. One can reasonably expect that a *deterministic* matching algorithm would need to perform more computation than such a nondeterministic

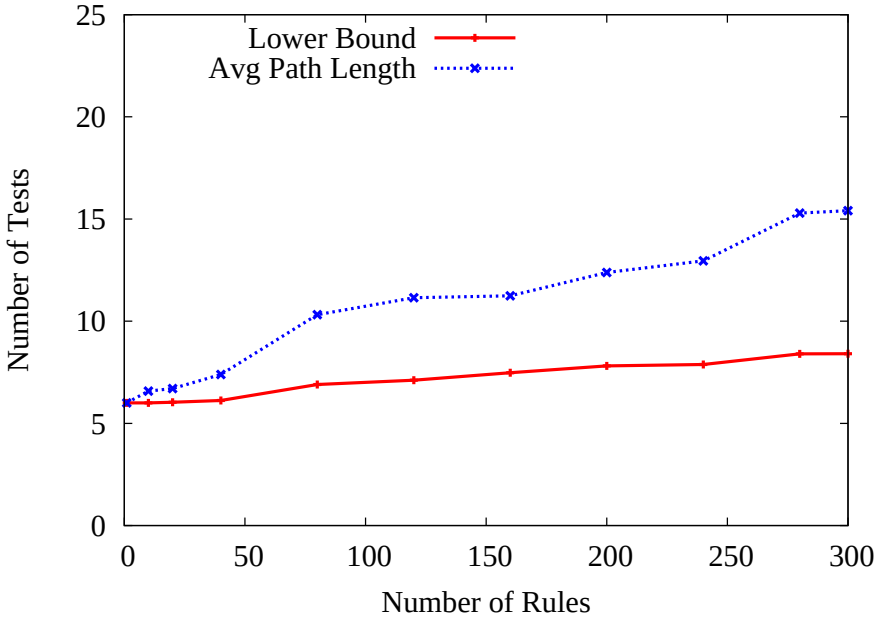

Figure 12: Path Length for Snort Rules

algorithm. For this reason, a deterministic algorithm that comes fairly close to the lower bound for nondeterministic algorithms, say, within a factor of two, could be considered a very good algorithm. We therefore use the ratio of actual matching cost to the lower bound for match verification cost as a metric for evaluating an automaton. In our experiments, we computed this metric statically: in particular, we computed the average of this ratio across all paths in the automaton.

Observation 7.1 (Minimum Match Verification Cost)

- *The lower bound for verifying a successful match of a filter F is $O(|F|)$.*
- *The lower bound for verifying a successful match of all filters in a set $\mathcal{M}$ is $O(k)$, where k is the number of distinct fields (or distinct number of field and bit-mask combinations) tested across all the filters in $\mathcal{M}$.*

It is clear that a match cannot be announced without testing all conditions in F and hence the bound in the first case. In the second case too, it is clear that all the fields present in all the filters in $\mathcal{M}$ have to be examined before announcing a match for all of them, and hence $O(k)$ time is needed.

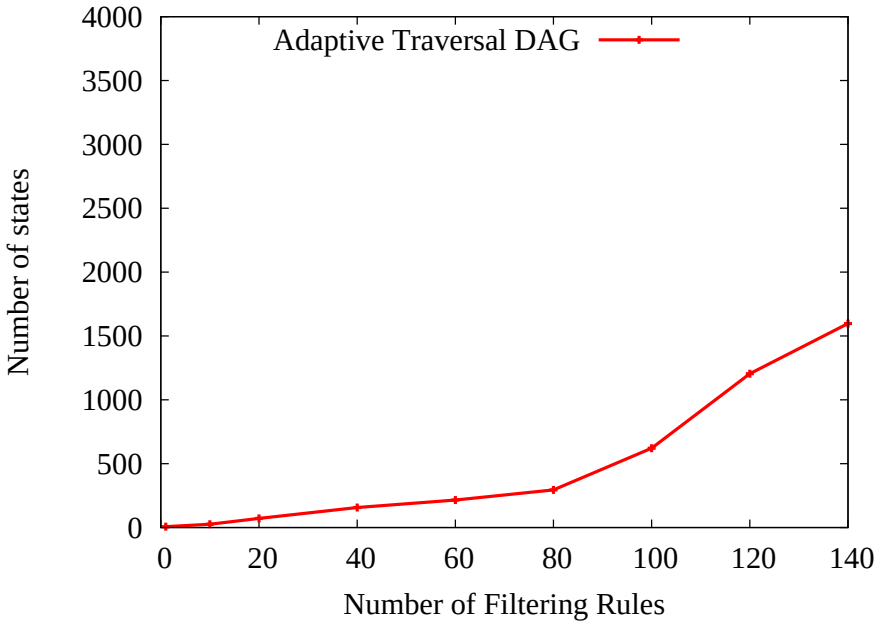

Figure 13: Automaton Size for Firewall Rules

In order to better understand the effectiveness of our technique in reducing the matching time, we compared the cost of our automata with the lower bounds for match verification cost in Figure 12. Our results show that our matching cost is within a factor of two from this bound. More remarkably, the number of tests performed increases by only a factor of 3 when the number of rules is increased from 1 to 300.

7.1.4 Experiments with Firewall Rules

The firewall rule set we considered is typical for a small to medium scale organization such as a department in a University. It divides a network into several subnets: the main network (all servers, workstations, etc), DMZ network, a wireless network, and a testbed network. The firewall is used for the traffic between these subnets and also between the outside world. The firewall rules are in the form of iptable rules for a Linux machine. There are different chains of rules for each of the subnets. Excluding the rules for defining and branching to user-defined chains, there were 140 actual filtering rules.

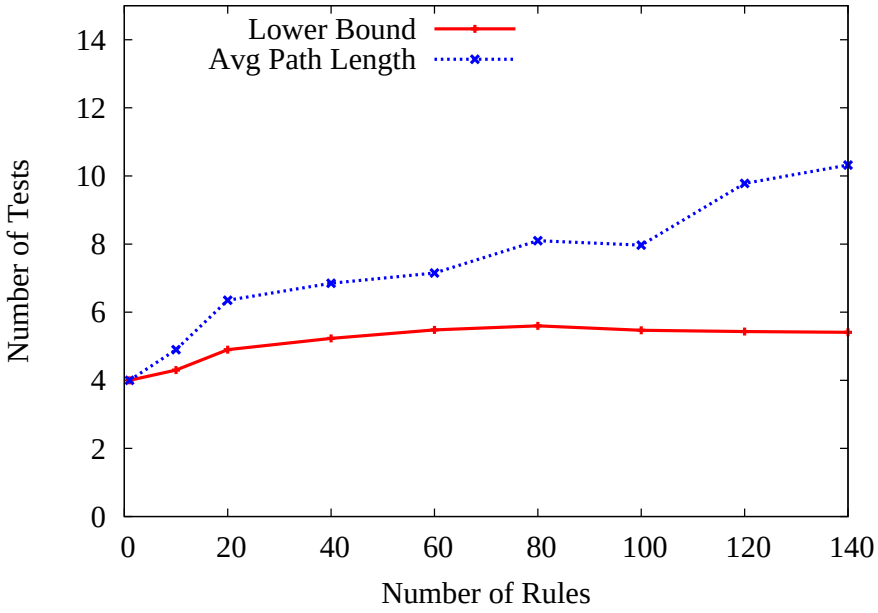

Figure 14: Matching Time for Firewall Rules

Figure 13 shows the automaton size as a function of the number of rules. The automaton size increases at a somewhat faster rate than in the case of network intrusion detection systems because firewall rules are totally ordered in terms of priorities. As a result, they can never have independent subsets of filters, and hence the benign nondeterminism technique cannot be applied.

Figure 14 compares the cost of our automata with the lower bounds for match verification. Although the results in this case seem similar to that obtained for network intrusion detection systems rules, we point out that they are actually better than what they appear to be. In particular, to verify a match for a filter F in the presence of priorities, it is not sufficient to just verify if the tests in F hold, but we also need to verify that at least one of the tests in each of the higher priority rules don't match. As a result, the match verification lower bound is strictly higher than the number for unprioritized rule sets that arise in the context of network intrusion detection systems.

Part II

Deep Packet Inspection

CHAPTER 8

Handling Content Matching

Modern intrusion detection systems use rules that look for patterns of known, suspicious activity in packet payload, in addition to tests on packet header fields. These systems typically scan evey byte of the packet payload and identify the matching patterns. This operation is called as *deep packet inspection* (DPI). In this chapter, we describe how we extend the techniques presented in previous chapters to handle DPI efficiently for network intrusion detection systems such as Snort.

8.1 Background in DPI

Snort rules consist of tests on packet header fields and the *strings* to be searched in the payload. Hence, rule matching in Snort involves matching the packet header fields and string matching operation over the packet payload. A Snort rule that specifies multiple strings can match only if all the strings in the rule are found in the payload. To reduce false positives, Snort allows rule writers to specify additional constraints on string matching. For example, a writer can specify constraints on strings s_1, s_2 in a rule such as, the rule matches only if:

- s_1 is found at a certain "offset" from the start of the payload,
- s_1 is found atleast "distance" bytes away from a previous match for s_2,
- s_1 is found "within" certain number of bytes from a previous match for s_2.

String matching is an expensive operation at it may require scanning every byte of the packet payload. Multi-pattern matching algorithms such as Aho-Corasick [1] and Wu-Manber [30] that identify all matching patterns in a single scan of the input can not be directly used in the presence of such constraints on strings. Moreover, due to these constraints, same input byte may need to be scanned multiple times. In particular, if a string is found but an associated constraint fails then the string matching has to be continued to find other occurrences of the same string. To see why this is needed, consider a rule that looks for a string "a" and another string "b", that is "within" 1 byte of "a". Now if a packet payload, contains "aab", then after the first "a" and the "b" is matched, the constraint fails. In this case, the string matching operations need to be performed again starting from the second byte for the rule to match. This slows down the string matching operation further and as a result the performance of rule matching deteriorates rapidly.

A straightforward way for improving the performance of rule matching is to avoid the slow string matching whenever possible. Snort uses an ad-hoc technique to achieve this. Snort uses a small set of packet fields that are used in almost all rules, e.g., source and destination ports (for TCP- and UDP-related rules), and type field (for ICMP-related rules), to divide the rules into different groups. For each such group, it extracts the longest string contained within the content-matching part of the rule, and builds an Aho-Corasick automaton for these rules. At runtime, a simple packet classification technique is used to identify the rule group against which a packet needs to be matched. Then the content of the packet is matched using the Aho-Corasick automaton associated with this group. Since this automaton only considers the longest string from each rule, some of the rules returned by this automaton may not really match the packet. (However, the automaton will always return a superset, not a subset of matching rules.) To handle the other complex conditions, e.g., a constraint on the distance between two strings within a rule, Snort performs a one-on-one match between a packet and each of the rules returned by the automaton. From here on, we will refer to these string operations which are performed on per rule basis as *slow search*.

8.2 Improving End-to-End Performance using Packet Classification Automata

Snort is able to efficiently construct these groups by limiting the number of fields used to divide the rules into groups. But the drawback of this approach is that the number of rules that belong to the same group remain substantial. As a result, the sequential matching phase can still take significant time.

We can improve this scheme, by replacing the simple packet classification scheme of Snort, with the matching automaton constructed by our technique. At each leaf of this automaton, we replicate the technique used by Snort, i.e., we build an Aho-Corasick automaton to recognize the longest string contained in each of the rules in the candidate set of the leaf. Finally, a one-on-one match is perfomed between the rules returned by this automaton and the network packet. The main benefit of this approach is apparent – by testing most of the packet fields first we ensure that number of sequential matches performed is substantially reduced. Section 8.5 shows the improvements that we achieve in the end-to-end performance of Snort using this approach.

We note here that in the presence of nondeterminism, we needed to modify the above technique so as to avoid repetition of string-matching tests after backtracking. Specifically, we built the Aho-Corasick at the first nondeterministic node encountered on a root-to-leaf path in the automaton, and performed an intersection of the set of rules returned by the Aho-Corasick with the rule sets of each of the matching leaves.

8.3 Incorporating String Matching in Packet Classification Automata

An even better approach to reducing the number of times the slow string matching operation is performed is to use all the strings tests directly in the matching automaton. In this section we describe how we extend the techniques presented in the previous sections to incorporate string matching tests.

To handle string matching, we extend the definition of tests (Definition 2.1) to include tests which look for strings in the payload. These tests are of the form

content = s_i, where s_i stands for each unique string that is tested in a rule set. Each filter may have multiple such tests. An example of a rule set, $\mathcal{F}$, containing such rules is:

- $F_1 : (tcp\_sport = 80) \wedge (content = \text{“}Command\ complete\text{”})$
- $F_2 : (tcp\_sport = 80) \wedge (content = \text{“}Bad\ command\text{”}) \wedge (content = \text{“}Bad\ filename\text{”})$
- $F_3 : (tcp\_sport = 25) \wedge (content = \text{“}Command\ complete\text{”})$

For ease of understanding, we represent each filter F_i as a conjunction of the packet field conditions C_i and the strings s_j that are tested by the filter. So the above filter set $\mathcal{F}$ can be succintly represented as:

- $F_1 : C_1 \wedge s_1$
- $F_2 : C_2 \wedge s_2 \wedge s_3$
- $F_3 : C_3 \wedge s_1$

where, s_1, s_2, and s_3 stand for *“Command complete”*, *“Bad command”*, and *“Bad filename”* respectively.

The obvious approach of directly incorporating string checks as tests in the automata does not work since each string test potentially requires the entire payload to be scanned. We overcome this problem by constructing an Aho-Corasick automaton from the strings and matching all the strings in a single scan of the payload. We use boolean variables to remember which strings matched in the Aho-Corasick automaton. We associate a boolean variable X_i with each s_i to indicate whether a string s_i is present in the payload of a packet. The values for these variables are bound when we run the Aho-Corasick algorithm on the packet payload. We can now replace each s_i in $\mathcal{F}$ with a test which checks whether X_i is set to get $\mathcal{F}'$:

- $F_1' : C_1 \wedge (X_1 = 1)$
- $F_2' : C_2 \wedge (X_2 = 1) \wedge (X_3 = 1)$
- $F_3' : C_3 \wedge (X_1 = 1)$

We can now construct a packet classification automata for $\mathcal{F}'$ treating the tests on these boolean variables just like tests on packet fields. At runtime, the values of

these boolean variables are bound by the string matching component. But to use the string matching component, we need to figure out the start of the payload in the packet. To ensure that we identify the type of the packet correctly before doing the string matching, we add preconditions (as explained in Section 6.1.3) to these boolean variables.

Even though this approach of utilizing all string tests reduces the number of slow searches performed as compared to the approach of the previous section, it may not result in significant runtime improvement. This is because in this approach we perform the Aho-Corasick matching, which involves scanning every byte of the payload, for every packet. We can avoid this by performing *lazy binding* for the boolean variables, i.e, performing the Aho-Corasick matching only when we reach a node that tests a boolean variable. Using this approach of late binding, we can avoid the Aho-Corasick matching for packets that match a path, from root to a leaf, that does not contain any such boolean variable test.

Since string matching is expensive we give preference to the simpler packet header fields. We modify *Select* function to pick these boolean variable tests at a state only when there are no remaining tests on packet fields. This ensures that the expensive string operation is performed for a packet only after all simpler packet header field tests have been performed.

Further improvements can be obtained by constructing separate Aho-Corasick automata for each path instead of constructing a single Aho-Corasick automaton that matches the strings in all the rules. The Aho-Corasick automaton for any path contains only the strings corresponding to the boolean variables tested on that path. In the presence of nondeterminism we modify this technique, as before, to construct an Aho-Corasick automaton at the first nondeterministic node on the path from the root-to-leaf. This automaton contains all the strings corresponding to all the boolean variables tested at the sub-tree rooted at this node.

8.4 Implementation

We generate the matching code for the automaton as explained in Section 6.1.3. For the nodes that test these boolean variables, we generate code which first checks if the values of the boolean variables are bound. If the variables are not bound, then the string matching is called using the Aho-Corasick automaton for that path,

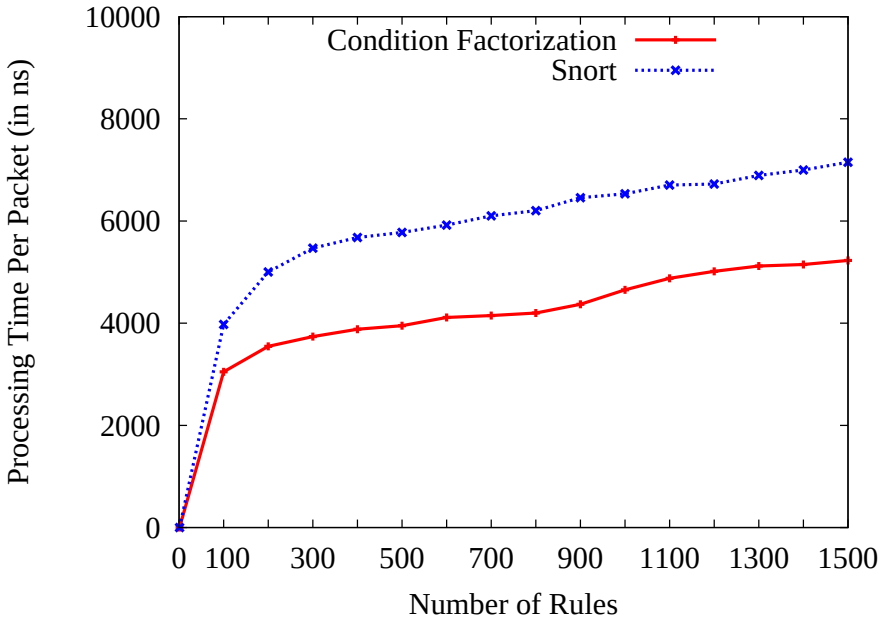

Figure 15: Total Matching Time

which binds the variables. From this point on the matching code can just check the values of these variables. For the rules matched by our automata, we perform the sequential match to identify the final matched rules.

We note that Snort supports writing patterns as regular expressions also. These regular expressions can be matched very fast by using deterministic finite-state automata (DFAs). But for several classes of rules DFAs can blowup in space. Hence, Snort uses nondeterministic finite-state automata (NFAs) to match regular expression at the expense of higher time complexity for rule matching. Separate NFAs are constructed for each rule and they are matched individually. We perform this regular expression matching operation as part of the final sequential match phase.

8.5 Evaluation of End-to-End Performance

In our experiment, we replaced the simple packet classification used in Snort 2.6 with the matching automata constructed by our technique. Our implementation reuses almost all of Snort code, including the code for Aho-Corasick automaton,

and the final one-on-one match. It only replaced the initial packet classification component. As a result, the performance improvements obtained by our technique are entirely due to the use of our sophisticated packet classifier.

We measured the total time taken by original Snort, and the version of Snort we modified to use our matching automaton. These times were computed for a 21-million packet trace collected in our laboratory consisting of about 30 hosts. We used the same rule set as in Section 7.1 but with all the tests now. So we had 1635 rules in this experiment.

In this experiment, we observed that the one-on-one matching phase was invoked about 120M times in the original Snort, whereas it was invoked only 40M times with our packet classifier in place. This reduction in the number of one-on-one matches translates to about 30% reduction in the overall time taken by Snort.

Figure 15 shows the overall time taken by Snort with and without our modification, as we vary the number of rules. While the performance is nearly identical for small rule sets, it quickly increases to (and stabilizes at) about 30% at a few hundred rules.

CHAPTER 9

Related Work

In this chapter, we discuss related research efforts in (i) packet header matching, and (ii) deep packet inspection.

9.1 Packet Header Matching

In this section, we discuss the related work in the area of packet header matching.

9.1.1 Early Works

The CMU/Stanford Packet Filter (CSPF)

The CMU/Stanford packet filter [19] is an interpreter based filtering mechanism. The filter specification language uses boolean expression tree. The tree model maps naturally into code for a stack machine. In the tree model, each interior node represents a boolean operation (e.g. AND, OR) while the leaves represent test predicates on packet fields. Each edge in the tree connects the operator(parent node) with its operand(child node). The algorithm for matching the packets proceeds in a bottom up manner. Packets are classified by evaluating the test predicates at the leaves first and then propagating the results up. A packet matches the filter if the root of the tree evaluates to true. Fig. 16 shows a tree model that recognizes either IP or ARP packet on Ethernet.

The major contribution of CSPF is the idea of putting a pseudo-machine language interpreter in the kernel. This approach forms the basis of many later-day

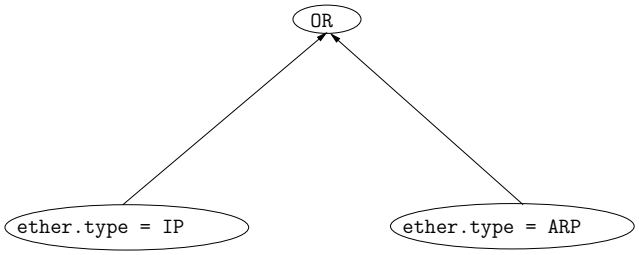

Figure 16: Tree Model

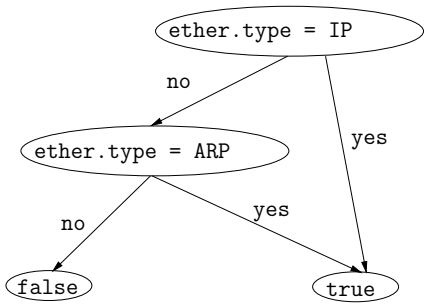

Figure 17: CFG Model

packet filter mechanisms. Also the filter model is completely protocol independent as CSPF treats a packet as a byte stream.

However, CSPF suffers from shortcomings of the tree model. The tree model of expression evaluation may involve redundant computations. For example, consider a filter that accepts all packets with an Internet adress `foo`. We want to consider IP, ARP, and RARP packets carried on Ethernet only. The tree filter function is as shown in figure 18. As can be seen the filter will compute the value of (`ether.type` = `IP`) even if (`ether.type` = `ARP`) is true. Although, this problem can be somewhat mitigated by adding *short circuit* operators to the filter machine, some inefficiency is inherent dure to the hierarchical design of network protocols. Packet headers must be parsed to reach successive layers of encapsulation. Since each leaf of the expression tree represents a packet field independent of other leaves, redundant parses ay be carried out to evaluate the entire tree.

There is also a performance penalty for simulating the operand stack. Moreover, the filter specification language is restricted to deal with only fixed length fields since it does not contain an indirection operator.

The BSD Packet Filter (BPF)

BPF [18] was originally created for BSD UNIX and has been ported to many UNIX flavors. It is also an interpreter based filter. It attempts to address some of the limitations of CSPF. BPF filters are specified in a low-level language. The language provides support for handling varying length fields. BPF uses directed acyclic control flow graph(CFG) model. In this model, each node node represents a packet field predicate. The edges represent control transfer. One branch is traversed if a

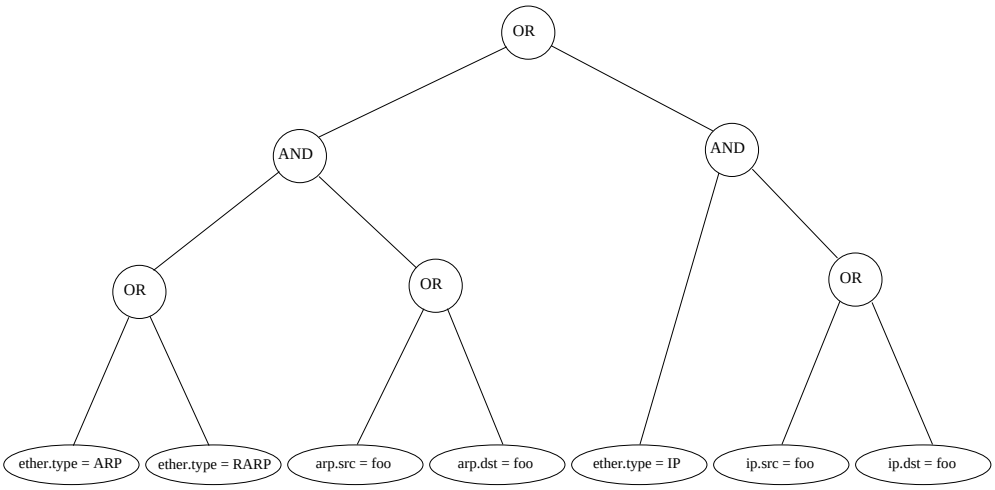

Figure 18: Tree filter for host `foo`

predicate is true and the other if it is false. Two terminating leaves represent true and false for the entire filter. The filter `IP` or `ARP` on `Ethernet` can be represented in CFG model as shown in fig. 17.

Use of CFG helps BPF to avoid some redundant computation. For example, the filter for accepting packets with an Internet address `foo` (as described in section 9.1.1) is represented in CFG model as shown in figure 19. However, BPF checks multiple filters sequentially and hence, does not scale well for large number of rules.

The Mach Packet Filter (MPF)

The Mach packet filter [32] enhances BPF to handle end-port based protocol processing in the Mach operating system. The primary focus of MPF is on demulplexing packets. So they consider only filters that share common prefix and differ at a single point in the header, say TCP port. This common prefix,recognized using simple template-matching heuristics, is merged and additional checks are included for the differing packet field. Although MPF performs demultiplexing efficiently, it does not provide a general way of matching multiple filters.

PathFinder

PathFinder [3] is a pattern based packet filtering mechanism that is designed so that it can be efficiently implemented in both software and hardware. It allows for

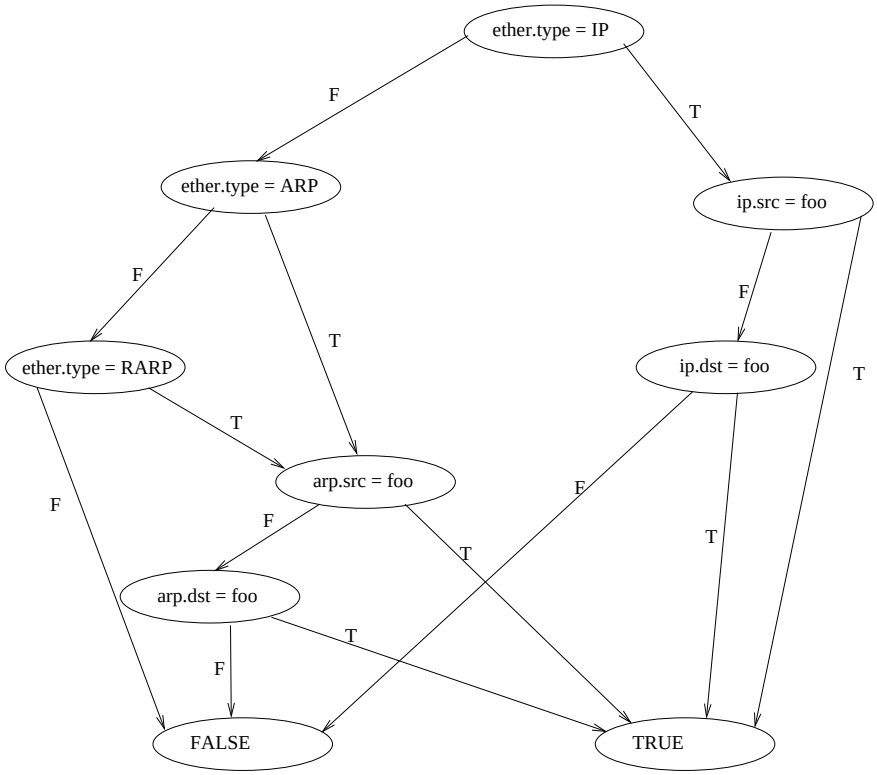

Figure 19: CFG Filter for host `foo`

more general composition of filters with common prefixes than MPF. The packet field predicates are represented by templates called "cells". The cells are chained together to form a "line". A line, which can be considered as a single filter, represents a logical `AND` operation over constituent predicates. A collection of lines i.e. a composition of filters, represents the logical `OR` operation over all lines. PathFinder eliminates common prefixes as new lines are installed. For example, filters for identifying two flows, say one from any source to destination `192.169.0.1 port A` and the other from any source to destination `192.169.0.1 port B` are composed as shown in figure 20. These optimizations only share tests that can be identified as common prefix of filters and hence, miss opporutnities to share other tests which do not occur as prefix.

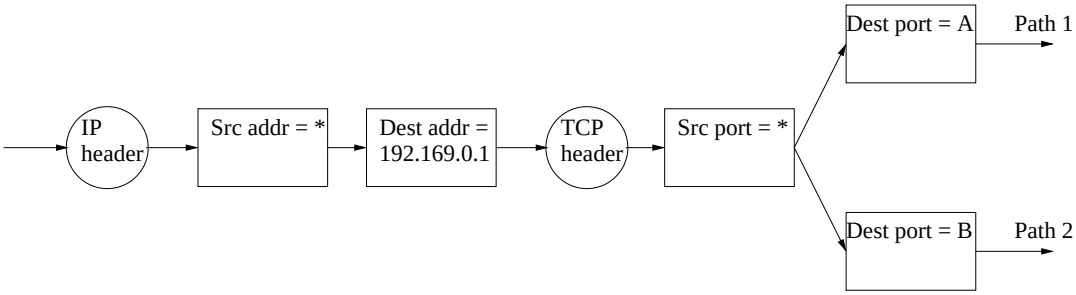

Figure 20: Composite filters in PathFinder

BPF+

BPF+ [4] provides a high-level declarative predicate language for representing filters. The BPF+ compiler translates the predicate language into an imperative, control flow graph. Before converting this control flow graph into low-level code, BPF+ applies a global data-flow algorithm, which they call as *redundant predicate elimination* for predicate optimization. BPF+ uses other common compiler optimizations like peep-hole transformations also. For example, if we specify a filter to accept all packets sent between `host X` and `host Y`, then a CFG representation would be as shown in figure 21. Here, MPF and PathFinder would not be able to perform any optimization as there is no common prefix. But BPF+ will be able to identify an opportunity for optimization using global data flow optimization techniques. If control reaches the node (`dest host` = `Y`) then we know that the source host is `X`. Therefore, the source host can not be `Y`. So the node (`source host` = `X`) is redundant. But this node can not be removed as there is another path through that node. So the dashed edge is transformed to point to `false` node. This reduces the average path length, and thereby improves filter execution performance.

All these optimizations are done while preserving the order in which the tests are specified. On the other hand, our technique reorders the tests to increase the opportunities for sharing common tests. Moreover, our condition factorization technique is more general than those of BPF+, being able to reason about semantic redundancies in the presence of bit-masking operations, and comparisons involving different constants.

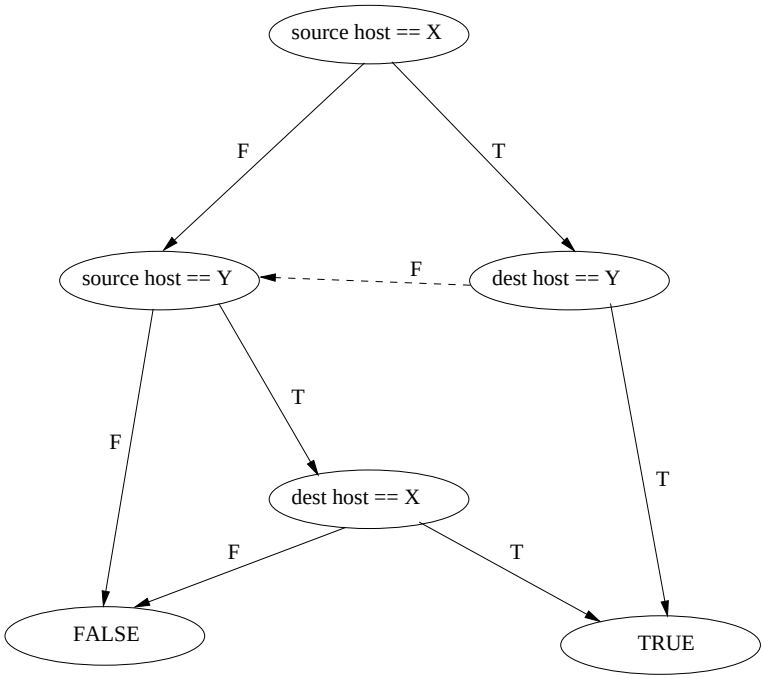

Figure 21: CFG for "all packets sent between X and Y"

9.1.2 Techniques targeted for routers

Many techniques for packet classification have been developed in the context of routers. These techniques restrict the problem so as to either work on a fixed number of fields or handle only certain forms of tests such as prefix lookup. Our techniques do not place any such restrictions. We discuss some of these techiques below.

Set-pruning tries

Decasper et al [6] present set-pruning tries that can be used where packet classification involves prefix matching for each field. Each field is considered as a set of bits that are checked in order. Longest prefix matching for a single field can be done efficiently by constructing a trie where each level in the trie corresponds to a bit being tested. Set-pruning tries extend this idea to handle multiple fields as follows. A trie is created for the first field. Leaves of this trie contain pointers to tries for the second field containing rules that match the first field. This process is repeated for other fields. Set-pruning tries suffer from memory blowup as rules can be duplicated

in the tries.

Grid-of-tries

Srinivasan et al [28] proposes grid-of-tries data structure for 2-dimensional classification, such as destination-source pairs, that avoids the memory blowup problem of set-pruning tries by allocating a rule to only one trie node. Grid-of-tries reduces query time by pre-computing and storing a *switch pointer* in some trie nodes. However, the performance of this technique degrades rapidly as the number of fields is increased.

Cross-producting

The paper presents another technique called *cross-producting* which performs better than grid-of-tries for more fields. Cross-producting works by creating lookup tables for each field. A packet is matched against each such table to identify the rules that match in that field. A final step that intersects the results for each field gives the rules that match a packet in all field. Cross-producting works only for prefix matching. Range checks are converted to (possibly multiple) prefix matches. Hence, the storage requirements for packet classification may increase rapidly when any field contains range checks. This technique can be used for only small rule sets because of the high worst-case storage requirement.

Tuple space search

Tuple space search [27] is another technique that addresses packet classification over multiple fields involving prefix matching. This technique creates a tuple for the number of bits that are checked in each field. For each such tuple, rules that check those bits are put in a set. For example, consider 2-dimensional rules $R_1 = \{10*, 110*\}$, $R_2 = \{110*, 1*\}$, and $R_3 = \{11*, 101*\}$. Here, R_1 and R_3 are put in the tuple set $(2, 3)$ while R_2 is put in tuple set $(3, 1)$. At runtime, each of these sets is searched using efficient techniques such as hashing. This technique works well only when the number of tuple sets is small.

HiCuts

HiCuts [10] partitions multi-dimensional search space guided by heuristics that exploit the structure of the packet classifiers. HiCuts builds a decision tree using local optimization at each node to choose the next field to test. The leaves of the tree contain some small number of rules that are tested in sequence.

HyperCuts

HyperCuts [25] reduces the depth of these trees by testing multiple fields at some levels. Hence, the tests at nodes are more expensive. They reduce this testing time by using arrays to identify the transitions. This results in increased storage requirements. These techniques suffer from exponential space requirement in the worst case.

9.1.3 Hardware based techniques

Many router vendors use Ternary CAMs that use parallel hardware to simultaneously check all rules. TCAMs are fast but unsuitable for large rule sets due to their power consumption and cost. Below we describe some algorithmic solutions that try to exploit hardware parallelism.

Bit-vector search

In the bit vector linear search algorithm [17], search is done for each field to yield sets of rules that match the packet for that particular field. These sets are intersected efficiently using bitmaps to yield the final set of rules that match the packet for all fields. This algorithm works well for moderate size rule sets.

Recursive Flow Classifier (RFC)

Recursive Flow Classifier [9] attempts to map all the bits of the packet header to an action identifier that corresponds to all the rules that can match the packet. It does this mapping over several stages. In each stage a set of memories, accessed in parallel, returns a value shorter (in terms of numer of bits used) than the index of the memory access. This technique also works well for moderate size rule sets as the memory requirement increases rapidly for larger rule sets.

9.1.4 Techniques based on reordering of tests

Dynamic reordering

Dynamic Packet Filter (DPF) [7] uses dynamic code generation, which allows dynamic reordering of tests. Dynamic reordering improves performance by detecting match failures earlier. Al-Shaer et al [12] and Woo [29] significantly improve on the dynamic reordering technique used in DPF by using efficient algorithms to maintain statistics regarding the traffic. Their techniques are analogous to profile-based optimizations in compilers, whereas ours is analogous to static-analysis based optimizations. Thus, the two techniques can complement each other.

Static reordering

Kruegel et al [14] build a decision tree using information gain to decide the order of testing fields. They present an approach for avoiding redundancy, where, by restricting the form of allowable tests, every test was converted into a canonical form so that semantically identical tests would also be syntactically identical. However, tests in canonical form can in general be more expensive than the original test, e.g., in order to support tests on IP addresses that may sometimes involve bit-masking operations and at other times involve equality, they convert both tests into smaller tests that examine one bit of address at a time. Moreover, the canonical form places more restrications on the form of tests.

9.1.5 Techniques from term rewriting

Pattern matching automata have been extensively studied in the context of term rewriting, functional and equational programming, theorem proving and rule-based systems. Augustsson described pattern matching techniques for functional programming that are based on left-to-right traversal [2]. Sekar et al [24] presented a technique for adapting the traversal order to reduce space and matching time complexity of term-matching automata. Gustafsson and Sagonas [11] extended this technique to handle binary data like network packets. Our technique generalizes their technique further by adding support for inequalities, disequalties, and bit-masks that are more general than their notion of bit-fields. More importantly, their automata has an exponential worst-cast space complexity. Although they describe a

technique for constructing linear-size *guarded sequential automata*, these automata require runtime operations to manipulate match and candidate sets. In particular, their transitions, strictly speaking, become $O(N)$ operations, which contrasts with our approach that takes $O(1)$ expected time per transition.

9.2 Deep Packet Inspection

Lot of research has focused on reducing the memory requirements of DFAs for intrusion detection system rules. Unlike our technique which tries to improve the end-to-end performance, these technique look at deep packet inspection in isolation. They focus on making regular expression matching more efficient. These techniques are complementary to the technique that we presented for improving the end-to-end performance.

Bro: Intrusion Detection System

Vern Paxson [20] developed Bro which is another popular network intrusion detection system. Bro has a powerful policy language that allows the use of sophisticated rules. Sommer and Paxson [26] enhanced Bro rule matching to use regular expressions. They build DFAs from the regular expressions. They overcome the problem of exponential space requirements of DFAs by building the DFAs incrementally at runtime. Moreover, they mention using the constraints on packet fields to reduce the size of the sets of rules that are compiled into a DFA. In that respect, our technique can help them in avoiding the exponential blowup, as we generate small sets for content-matching.

Multiple DFAs

Fang Yu et al [31] studied the regular expressions commonly used in network monitoring and security applications. They identified the features in these regular expressions that cause exponential blowup when they are compiled into DFA. They propose regular expression rewriting techniques to reduce memory usage. They use a set-splitting technique that combines rules into multiple DFAs instead of a single DFA. They use simple heuristics to decide which rules should be grouped together

to stay within available memory budget. Matching in this case may require the traversal of multiple automata.

Delayed Input DFA (D2FA)

Delayed Input DFAs [15] reduce the number of transitions stored in each state. Each transition table stores only the transitions that are distinct to that state. Transitions that are common to many states are stored in a transition table that can be reached by default transition in other states. This approach may require the traversal of multiple default transitions per byte.

Multiple Alphabet Compression Tables

Estan et. al [13] use an orthogonal approach to reduce the number of transitions. They use multiple alphabet compression tables to reduce the number of entries in the transition table of each state. This approach adds one extra lookup per input byte.

CHAPTER 10

Conclusions and Future Work

10.1 Conclusions

In this dissertation we presented a new technique for fast packet classification. Unlike previous techniques, our technique is flexible enough to support filtering as well as classification applications. It can support prioritized rules such as those used in firewalls, as well as unprioritized rules requiring all matches to be reported, such as those used in intrusion detection systems. We developed novel techniques and algorithms that guarantee polynomial size automata, while, in practice, avoiding any repetition of tests. Our experimental results show that the technique is very effective in reducing automata size as well as matching time. Finally, we presented techniques for combining string matching and packet classification that can be used to achieve high end-to-end performance for rule matching.

10.2 Future Work

There is a lot of interesting work that still needs to be done. This includes:

- In Part II of this dissertation, we presented techniques for incorporating string matching tests in our automata construction algorithm. In future we want to extend these ideas to handle the constraints on string matching (Section 8.1). We can associate some variables with these constraints similar to the boolean variables associated with string tests. For example, to handle the

constraint that a string match occurs within certain *depth*, say `100` bytes of the payload, we can use a test (`depth` $\leq$ `100`). The value of `depth` is bound when a string match occurs. Such tests can be directly used in the automaton construction. To ensure the correctness of these tests, the test on boolean variable associated with the string test for the constraint, is added as precondition. As explained before, the *select* implementation will ensure that the preconditions are satisfied before the test. We plan to extend our technique to incorporate all tests which can be computed by storing some variables at the states in the Aho-Corasick automata.

- We can increase the opportunities for sharing of string match tests by using a different encoding for the variables associated with strings. One possible way to do this is to associate the same variable with multiple strings that have certain parts in common. For example, if a string s_1 is a substring of s_2, then we can associate a variable X_1 with both. We can then set X_1 to 1 if s_1 is found and to 2 if s_2 is found. Now we can associate test $(X_1 \geq 1)$ to stand for finding s_1 and $(X_1 \geq 2)$ for finding s_2. Now, using the residue computation of section 3.1.1, it is clear that $(X_1 \geq 1)/(X_1 \geq 2)$ is *true*. This captures the notion that if string s_2 is found, then that implies that s_1 is also present. We plan to investigate other encodings such as bit-mask based tests to increase the opportunities for sharing the results of string tests.

- Recent years have seen attackers targeting the vulnerabilities in higher layer applications. For instance, newer attacks target the applications running on the web server rather than the web server itself. *Vulnerability signatures*, which are predicates on higher level application fields, are more effective at detecting such attacks than simple string or regular expression based signatures. Matching vulnerability signatures is very expensive as it requires complex tasks such as application parsing and session maintenance. Researchers are working on improving the performance of these tasks. Another hindrance for the widespread adoption of vulnerability signatures is that current techniques for rely on sequential search. Our technique can improve the performance of this phase. We can use the packet classification automaton to match the application level fields on behalf of multiple signatures. As before, preconditions on these fields can be used to enforce any constraint on the relative

order of testing these fields. Preconditions can also be used when the parsing of a field depends on the value of an earlier field.

Bibliography

[1] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–343, 1975.

[2] L. Augustsson. Compiling pattern matching. *Functional Programming and Computer Architecture*, 1985.

[3] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. Pathfinder: A pattern-based packet classifier. In *Operating Systems Design and Implementation*, 1994.

[4] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *SIGCOMM*, 1999.

[5] S. Chandra and P. McCann. Packet types. In *Second Workshop on Compiler Support for Systems Software (WCSSS)*, 1999.

[6] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: a software architecture for next generation routers. *SIGCOMM Computer Communication Review*, 28(4):229–240, 1998.

[7] D. R. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM*, 1996.

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

[9] P. Gupta and N. McKeown. Packet classification on multiple fields. In *SIGCOMM*, 1999.

[10] P. Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects*, 1999.

[11] P. Gustafsson and K. Sagonas. Efficient manipulation of binary data using pattern matching. *J. Funct. Program.*, 16(1):35–74, 2006.

[12] H. Hamed, A. El-Atawy, and E. Al-Shaer. On dynamic optimization of packet matching in high-speed firewalls. *IEEE Journal on Selected Areas in Communications*, 24(10), 2006.

[13] S. Kong, R. Smith, and C. Estan. Efficient signature matching with multiple alphabet compression tables. In *SecureComm*, 2008.

[14] C. Kruegel and T. Toth. Using decision trees to improve signature-based intrusion detection. In *Recent Advances in Intrusion Detection*, 2003.

[15] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM*, 2006.

[16] MIT Lincoln Labs. Darpa intrusion detection evaluation. http://www.ll.mit.edu/IST/ideval, 1999.

[17] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *SIGCOMM*, 1998.

[18] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, 1993.

[19] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Symposium on Operating Systems Principles*, 1987.

[20] V. Paxson. Bro: A system for detecting network intruders in real-time. In *USENIX Security*, 1998.

[21] R. Ramesh, I. Ramakrishnan, and D. Warren. Automata-driven indexing of prolog clauses. In *Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 281–290, San Francisco, 1990. Revised version appears in Journal of Logic Programming, May 1995.

[22] M. Roesch. Snort - lightweight intrusion detection for networks. In *Systems Administration Conference, USENIX*, 1999.

[23] R. Sekar, Y. Guang, S. Verma, and T. Shanbhag. A high-performance network intrusion detection system. In *ACM Conference on Computer and Communications Security*, 1999.

[24] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. In *SIAM Journal of Computing*, pages 1207–1234, 1995.

[25] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *SIGCOMM*, 2003.

[26] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *ACM Conference on Computer and Communications Security*, 2003.

[27] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple search space. In *SIGCOMM*, 1999.

[28] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *SIGCOMM*, 1998.

[29] T. Y. C. Woo. A modular approach to packet classification: Algorithms and results. In *INFOCOM*, 2000.

[30] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, 1994.

[31] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Architectures for Networking and Communications Systems*, 2006.

[32] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *USENIX Winter*, 1994.