Fast Pattern-Matching Techniques for Packet Filtering

by

Alok S. Tongaonkar

to

The Graduate School
in partial fulfillment of the
Requirements
for the degree of

Master of Science
in
Computer Science

Stony Brook University
May 2004

Stony Brook University

The Graduate School

Alok S. Tongaonkar

We, the thesis committee for the above candidate for the

Master of Science degree,

hereby recommend acceptance of this thesis.

Professor R. C. Sekar, Thesis Advisor,
Computer Science Department

Professor I. V. Ramakrishnan, Chairman of Thesis Committee,
Computer Science Department

Professor C. R. Ramakrishnan,
Computer Science Department

This thesis is accepted by the Graduate School.

Dean of the Graduate School

Abstract of the Thesis

Fast Pattern-Matching Techniques for Packet Filtering

by

Alok S. Tongaonkar
Master of Science
in
Computer Science
Stony Brook University
2004

Packet filtering is used for selecting or classifying network packets in a variety of network applications such as routers and network monitors. Packet filters are typically specified using patterns. These patterns specify constraints on the values of different fields in the packets . The two key requirements in packet filtering are *high performance* and *flexibility.* High performance refers to the ability of the packet filtering system to quickly compare a packet with different patterns. Flexibility refers to the ability of the system to be easily applied for different filtering applications.

Pattern matching is a well studied problem that has applications in various fields such as functional programming and rule-based systems. In this thesis we generalize and apply techniques from pattern matching to develop high performance packet filtering systems that can be used in a variety of applications such as intrusion detection systems and network monitors. The algorithm aims to minimize the matching time and space requirements of the generated packet filtering system. These factors are crucial in applications that are charecterized by large number of filters such as intrusion detection system.

We discuss the implementation of this algorithm and present experimental results to demonstrate the scalability and performance of the algorithm.

Table of Contents

List of Figures

ACKNOWLEDGEMENTS

I am grateful to take this opportunity to sincerely thank my thesis advisor, Dr. R.C. Sekar, for his constant support, invaluable guidance and infinite patience.

I wish to extend my sincere thanks to Dr. I.V. Ramakrishnan and Dr. C.R. Ramakrishnan for consenting to be on my defence committee and offering valuable suggestions.

I would like to thank my friends: Mohan-Krishna Channa-Reddy for helping me throughout my thesis research; Salil Gokhale for his insightful suggestions at various times; Amit Purohit and Sandeep Bhtakar for their constant encouragements; Akshay Athalye, Sumukh Shevde and Gaurav Gothoskar for making my stay at Stony Brook a pleasant time; V.N. Venkatakrishnan for his invaluable suggestions for the thesis defense; Kiran Reddy for helping me in preparing the final version of the thesis.

Finally, I would like to thank my parents, sister Meghana and brother-in-law Nitin Sadawarte, 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

Packet filtering is a mechanism that inspects incoming network packets, and based on the values found in select fields, determines how each packet is to be processed. Packet filtering is used in many applications like network monitoring, performance measurement, demultiplexing end-points in communication protocols, packet classification in routers, firewalls, and in intrusion detection.

Typically, these applications specify multiple filters that are to be applied to a network packet. These specifications can be in the form of imperative code written using some special-purpose filtering language. The code for a packet filter consists of tests that are to be performed on the packet fields and the action to be taken when the fields of a packet pass all the tests of the filter. The code for multiple filters consists of codes for individual filters that are combined using appropriate control flow instructions. Clearly, the size of the code increases as the number of filters increases. Writing such filter code manually is cumbersome and generally error-prone. Adding a filter or changing the protocol requires significant rework in the case of such code. Hence, maintaining the filtering code becomes difficult.

In this thesis we follow a different approach for specifying the filters. In this approach, a filter specification is in the form of a pattern that specifies constraints on the values of different fields in the packet. A packet matches a pattern if the packet fields satisfy the constraints specified in the pattern.

A packet filtering system containing multiple filters can match a packet against the patterns in two ways. In the first approach, each filter is considered as separate entity. The filtering system runs each filter sequentially on every network packet. Here, each packet is matched against every pattern without using information gained about packet fields from previous match. The cost of identifying the matching pattern(s) grows linearly with the number of filters specified. This cost is unacceptable in applications that specify a large number of filters. Figure 1.1 shows an application like intrusion detection in which a number of filters are specified as

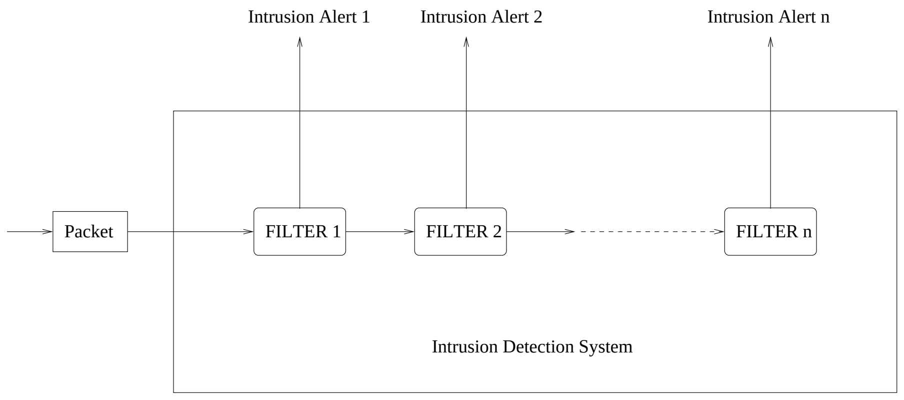

Figure 1.1: Individual Filters

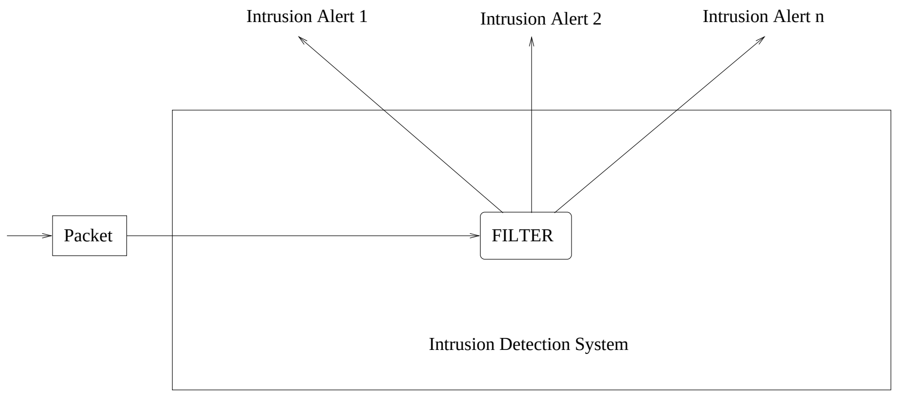

Figure 1.2: Composite Filters

patterns. Here each pattern characterizes a different network intrusion. A packet filtering system that tries to match each network packet against every pattern acts as a bottleneck to network performance.

In the other approach, a *composite filter* is generated by combining indiviual filter specifications as shown in figure 1.2. The advantage of this approach is that it tries to use information gleamed about packet fields from each partially successful match in the subsequent matches. Minimizing the space and matching time requirements of a composite filter is a key issue in filter composition.

A sequence of patterns can be compiled into a finite tree automaton that identifies the matching pattern efficiently. In a finite tree automaton each state, except the final states, tests *attributes* of network packets. There is an edge for each test. Each state can have a "default" edge which is used when all other tests fail in that state. The final states represent the patterns that are matched. The automaton can be either nondeterministic or deterministic. Traversal of the nondeterministic automaton will involve backtracking. So simulation of a nondeterministic automaton at runtime can be inefficient. Hence, a deterministic automaton is preferred over its nondeterministic counterpart. In the deterministic automaton each attribute is checked only once. So the running time is better than that of the corresponding nondeterministic automaton. But the size of a deterministic automaton can be far too large for it to be a practical solution.

A composite filter, represented as a deterministic automaton, can be viewed as a *decision tree*. A decision tree is a tree-like representation of a finite set of if-then-else rules. Each node of a decision tree is either a *decision node* or a *leaf node*. A decision node specifies some test to be carried out on a single attribute value, with one branch and sub-tree for each possible outcome of the test. A leaf node indicates the patterns that are matched. A set of matching patterns can be considered as a class for the packet. A decision tree can be used to classify a packet by starting at the root and traversing down to a leaf node, which provides the matching patterns.

For example, consider that a network packet has three fields: $f1$, $f2$, and $f3$. We want to specify four packet filters with the following patterns:

$p1 : (f1 == a) \&\& (f2 == b)$

$p2 : (f1 == a)$

$p3 : (f1 == c) \&\& (f3 == d)$

$p4 : (f2 == b)$

Figure 1.3 shows a decision tree automaton for classifying a network packet using the above patterns. In this case, a packet can belong to one of the five possible classes. These classes are

- Class 1: $\{p1, p2, p4\}$
- Class 2: $\{p3, p4\}$
- Class 3: $\{p2\}$
- Class 4: $\{p3\}$
- Class 5: $\{p4\}$

In such a decision tree, matching time for a packet belonging to a particular class is equal to the depth of the leaf node containing that class. The set of possible classes is a subset of the power set of the set of patterns specified. So the size of the decision tree can be exponential in the number of patterns specified. Minimization of the matching time and size requirements of deterministic tree automata has been studied in pattern matching context for applications like term rewriting and functional programming. However, since the problem of finding the optimal decision tree is NP-complete [5], heuristics are used to minimize the automaton. The heuristics,

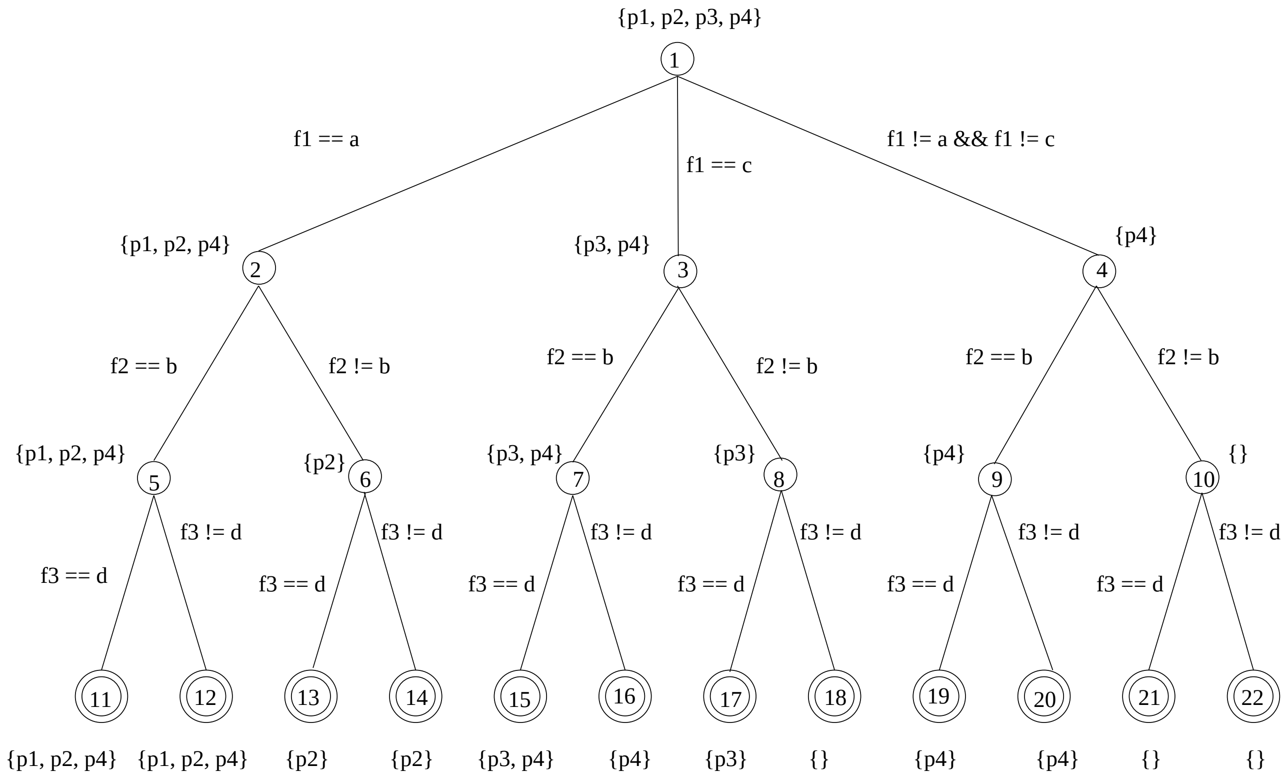

Figure 1.3: Deterministic Automaton

based on pattern matching techniques, try to minimize the automaton by sharing common tests.

The earliest packet filter mechanisms, CMU/Stanford Packet Filter (CSPF) [8] and Berkeley Packet Filter (BPF) [7] are interpreter-based filter mechanisms that do not support composition. Mach Packet Filter (MPF) [11] extended BPF to add support for composing filter specifications for special cases. MPF merges the common prefix in different patterns. PATHFINDER [1] is a pattern based packet filtering mechanism that allows for more general composition of filters with common prefixes than MPF. Dynamic Packet Filter (DPF) [4] enhanced PATHFINDER's core model with dynamic-code generation to improve performance. Jayaram and Cytron [6] explored a new approach of specifying each filter by a context-free grammar. This approach simplifies filter composition. Finally, BPF+ [2] allows multiple packet filters to be specified in a high-level language and be compiled into native-code. BPF+ performs low-level data flow optimizations like redundant predicate elimination to improve the performance of the packet filtering code.

All these works focus on exploiting common tests while preserving the order in which the tests are performed. So the extent of sharing possible is dependent on the order in which the tests are specified. In this thesis, we present a new approach which increases the extent of sharing. Our approach, based on the *Adaptive Pattern Matching* [10] technique, uses heuristics to *adapt* the traversal order to suit the input patterns. The modification of the order in which tests are performed increases the opportunities for sharing. This helps to further minimize the space and matching time requirements of the automaton.

The goal of this work is to generalize and extend pattern matching techniques to network packet filtering, and investigate their effectiveness. Pattern matching techniques consider only those tests which check whether an attribute has a particular value. Packet filters, on the other hand, commonly contain tests which check whether an attribute has a value in a particular interval. So a key challenge to adapting these techniques to network packet filtering is to handle intervals efficiently. Packet filters contain tests involving disequalities also. So the pattern matching techniques need to be extended to handle such tests. They also need some mechanism to handle bit-masking operations which are used commonly in packet filters. Another important consideration in network packet filtering is that the same bits in the packet may be interpreted in different ways. For example, a filter may contain a test involving bit-masking operation on certain bits and some other test which views the same bits as an integer.

In this thesis we have developed an algorithm, based on the adaptive pattern matching technique, to generate packet filters from high level specifications. The algorithm aims to determine a suitable order of testing attributes of network packets based on the patterns specified.

1.1 Thesis Overview

In the next chapter we see the language for specifying the filters. Chapter 3 explains the pattern-matching algorithm in detail. We give the summary in chapter 4.

Chapter 2

Background

In this chapter we review the languages available for specifying packet filters and the existing packet filter schemes .

2.1 Languages For Packet Filtering

In this section we review some of the special-purpose languages for specifying packet filters. The choices range from low-level imperative languages like the one provided by BPF [7], a high-level packet specification language like PacketTypes [3], to a high-level declarative language such as Behavioral Monitoring Specification Language (BMSL) [9].

2.1.1 BPF

BPF uses a pseudo-machine language interpreter to provide the abstraction for describing and implementing the filtering mechanism. The BPF machine abstraction consists of an accumulator, an index register (x), a scratch memory store, and an implicit program counter. BPF treates a packet as byte array. A packet field is accessed by accessing the bytes at the corresponding offset in the byte stream. Packet field values can consist of 32-bit words, 16-bit unsigned halfwords, and 8-bit unsigned bytes. The following memory access, data manipulation, and control flow instructions are provided:

- *Load Instructions* for copying values into the accumulator or index register. The source can be an immediate value, packet data at a fixed data, packet data at a variable offset, the packet length, or the scratch memory store.
- *Store Instructions* for copying either the accumulator or index register into the scatch memory store.

- *ALU Instructions* for performing arithmetic or logic on the accumulator using the index register or a constant as an operand.
- *Branch Instructions* for altering the flow of control , based on comaprison test between a constant or x register and the accumulator.
- *Return Instructions* for terminating the filter.
- *Miscellaneous Instructions* for register transfers.

Each filter is expressed using code that terminates with either an indication of what portion of the packet to save or a zero. A return value of 0 corresponds to the rejection of the packet by the filter. Acceptance of a packet is indicated by a non-zero return value.

Consider TCP/IP packets carried on Ethernet. A filter for accepting all IP packets is specified as:

```
       ldh   [12]
       jeq   #ETHERTYPE_IP, L1, L2
L1:    ret   #TRUE
L2:    ret   #0
```

The first instruction loads the Ethernet type field, which is an unsigned halfword at an offset of 12 bytes from the start of the packet, into the accumulator. This field is compared with type IP. If the comparison fails, then the packet is rejected and zero is returned. Otherwise,the packet is accepted and TRUE is returned. TRUE is a non-zero value that indicates the number of bytes of the packet that are saved.

A filter for rejecting all packets originating from IP networks, 128.3.112 or 128.3.254 is specified as:

```
       ldh   [12]
       jeq   #ETHERTYPE_IP, L1, L4
L1:    ld    [26]
       and   #0xffffff00
       jeq   #0x80037000, L4, L2
L2:    jeq   #0x8003fe00, L4, L3
L3:    ret   #TRUE
L4:    ret   #0
```

Similar filters can be written for other network protocols or media. BPF achieves this protocol independence by treating a packet as a simple array of bytes. But there is a flip side to this approach of viewing a packet as a sequence of bytes. The application that specifies the filters is responsible for encoding the filter appropriately for the underlying network media and protocols. In the case of multiple filters, writing such low-level code becomes cumbersome, reducing the usability of the language. A more serious drawback of this style of programming, where the layout of a network packet is explicitly encoded in a low-level language, is that it may introduce coding errors in offsets, sizes, and conditionals. Errors like accesing an offset that is outside the packet boundaries may cause a memory protection fault. Semantic errors like accessing an offset believing that it contains certain information, when in fact, the packet may be of a totally different type and contain completely different information, may arise even more frequently. Language features that increase usability and minimize the likelihood of these common errors are needed.

The usability of packet filters is increased by providing a user-friendly, high-level filter specification language. The high level filter specifications are translated into low-level code like BPF programs. Typically, this translation introduces redundancy in the filter programs. Previous works like PATHFINDER and BPF+ have tried to eliminate this redundacy by using various heuristics.

The problem of errors can be dealt with by hand-crafting a type checker that is developed explicitly for a prespecified set of network protocols. This approach, used in *tcpdump*, hard-codes the structures of packets for the prespecified protocols into the compiler, which is used for translating the high-level filter specifications into low-level filter code. This approach requires a redesign of the compiler to accomodate protocols that are not already built into the compiler.

2.1.2 PacketTypes

PacketTypes is a high-level packet specification language that is used to describe packet formats. Unlike other high-level filter specification languages, this language does not hardcode the protocol specifics into the compiler. Instead, the language provides a type system to specify the structure and content of the packets. This approach makes it easy to support new protocols. Another feature of the language is that the fundamental operation on packets is checking their membership in a type. Hence, type definitions written using PacketTypes can serve as packet filter specifications.

We take a look at how PacketTypes can be used to specify packet filters. Let us consider `TCP/IP` protocol. An IP packet might be specified as

```
nybble : = bit[4];
short := bit[16];
ipaddress := byte[4];
ipoptions := bytestring;

IP_PDU := {
  nybble      version;
  nybble      ihl;
  byte        tos;
  short       totallength;
  ...
  ipaddress   src;
  ipaddress   dest;
  ipoptions   options;
  bytestring  payload;
} ...
```

Here, IP_PDU defines the fields of IPv4 header. Although this type imposes a structure on packets, if no additional constraints are specified then it allows many bit sequences that are not valid IP packets. The necessary constraints like requiring the *version* value in IP_PDU to be 4, appear in a *where* clause following the sequence, as in:

```
IP_PDU := {
  ...
} where {
  version#value = 0x04;
  ...
}
```

The where clause can contain *demultiplexing constraints*, which are constraints comparing the value of a key field to a constant. These demultiplexing constraints can be used as filtering conditions. For example, a filter for capturing IPv4 packets originating from 192.169.0.1 is specified as:

```
IP_PDU := {
  ...
} where {
  version#value = 0x04;
  ...
  src#value = 192.169.0.1;
}
```

The language captures the layering of protocols by providing a construct called as *refinement*. Refinement is represented by the :> operator. Refinement uses constraints to augment the traditional notion of inheritance. For example, an IP packet on Ethernet might be specified as

```
macaddr := bit[48];

Ethernet_PDU := {
  macaddr     dest;
  macaddr     src;
  short       type;
  bytestring  payload;
}

IPinEthernet :> Ethernet_PDU where {
  type#value = 0x0800;
  overlay payload with IP_PDU;
}
```

Ethernet_PDU contains the specification of an Ethernet frame and IPinEthernet shows how to layer IP on it by constraining the type to have the value 0x800 and overlaying the IP_PDU definition onto the Ethernet payload.

Multiple filters can be specified by refining an existing type with different demultiplexing constraints. Filters for capturing packets originating from either 128.3.112.1 or 128.3.112.2 are specified as:

```
Filter1 :> IPinEthernet where {
  payload.srcaddr#value = 128.3.112.1;
}

Filter2 :> IPinEthernet where {
  payload.srcaddr#value = 128.3.112.2;
}
```

2.1.3 BMSL

BMSL is a pattern specification language that is developed at Secure Systems Lab at Stony Brook University. It can be used to specify packet filters. It provides a type system, similar to the one provided by PacketTypes, for specifying the structure and content of the packets . It has a notion of inheritance which is similar to type refinement in PacketTypes.

In spite of the similarities there are several significant differences between the two approaches. The inheritance mechanism of PacketTypes offers more power than that of BMSL in that it can capture protocols that use trailers also. BMSL trades off this power for simplicity.

We used BMSL for specifying the packet filters.

2.2 BMSL - A Pattern Specification Language

This section describes the features of BMSL and how it can be used for specifying packet filters.

2.2.1 Syntax

Specifications consist of variable and type declarations, followed by a list of rules [9]. The rules are of the form

```
pat  --> action.
```

Pat is a pattern on sequences of network packets. *Action* consists of a sequence of statements that are executed when there is a match for *pat*. If multiple patterns match at the same time, actions associated with each pattern are launched. The interaction of these actions that are launched simultaneously is beyond the scope of this thesis.

2.2.2 Event Declaration

The language supports declaration of *events*. These events may be *primitive* or *user-defined*. The primitive events are generated by external system and form the input to the generated packet filtering system. For example, there may be two events(say, tx and rx), corresponding to the transmission and reception of packets. These events may have a packet as an argument. They may have an additional argument that specifies the interface.

```
event rx(int interface, ether_hdr p);
event tx(int interface, ether_hdr p);
```

User-defined events, also called *abstract* events, correspond to the occurrence of a sequence of primitive events. They are declared as

```
event eventname{parameter1, parameter2, ..., parametern) = pat
```

where *pat* is an event pattern. Event patterns are described in Section 2.2.5.

2.2.3 Packet Structure Description

The structure of the packets can be specified using packet type declarations. The syntax of type declaration for packets is similar to that of the C-language. For example, the following describes an Ethernet header.

```
#define ETHER_LEN 6
struct ether_hdr {
  byte  e_dst[ETHER_LEN];   /* Ethernet destination address */
  byte  e_src1[ETHER_LEN];  /* Ethernet source address */
  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. BMSL permits multilevel inheritance to capture protocol layering. BMSL augments inheritance 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 */
  short        tot_len;    /* Total Length */
  ...
  short        check_sum;  /* Header Checksum */
  unsigned int s_addr;     /* Source IP Address Bytes */
  unsigned int d_addr;     /* Destination IP Address Bytes */
};
```

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.

2.2.4 Constraint Checking

An important requirement for the language to be type safe is that the constraints must hold before the fields corresponding to a derived type are accessed. Note that at compile time the actual type of the packet is not known. For example, a packet on an Ethernet interface must have the header given by `ether_hdr`. But it is not known whether the packet carries an ARP or an IP packet. So the constraint associated with `ip_hdr` must be checked at runtime before accessing the IP-relevant fields. Similarly, before accessing TCP relevant fields, the constraints on `tcp_hdr` must be checked. Furthermore, the constraints on `ip_hdr` must be checked before checking constraints on `tcp_hdr`.

2.2.5 Patterns

Patterns on packet sequences typically consist of a single receive or transmit event along with a condition. A pattern is said to be matched when the event specified in the pattern occurs and the condition associated with the event is satisfied. The condition denotes a boolean-valued expression involving the event arguments and possibly other variables. As we have seen in Section 2.2.2, a packet may be an event argument. Hence, the condition may include predicates involving packet fields.

Suppose we want to write a packet filter that captures all packets coming from host "foo" whose IP address is "xx.yy.zz.ww". Then we can write,

```
rx(ifc, p) | (p.s_addr  == xx.yy.zz.ww)
       --> { display(‘‘foo’’); };
```

The compiler for BMSL will automatically insert the constraint for `ip_hdr` in the condition.

```
rx(ifc, p) | (p.e_type == ETHER_IP) && (p.s_addr  == xx.yy.zz.ww)
       --> { display(‘‘foo’’); };
```

Multiple filters are specified as different patterns. Different predicates can be combined using boolean operators like AND, OR, and NOT. The compiler supports ranges and arithmetic and logic operations on packet fields as well as other variables.

Notice that there is a rule for each filter and each rule has a pattern. So from now on we will use the terms filter, rule, and pattern interchangeably.

2.3 Related Work In Packet Filters

In this section we describe the related work in the area of packet filters. We discuss six major packet filter schemes: CSPF [8], BPF [7], MPF [11], PATHFINDER [1], DPF [4], and BPF+ [2].

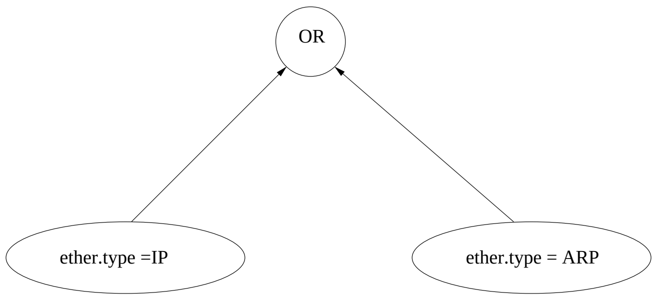

Figure 2.1: Tree Model

The CMU/Stanford Packet Filter(CSPF)

The CMU/Stanford packet filter 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. 2.1 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 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 2.2. 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.

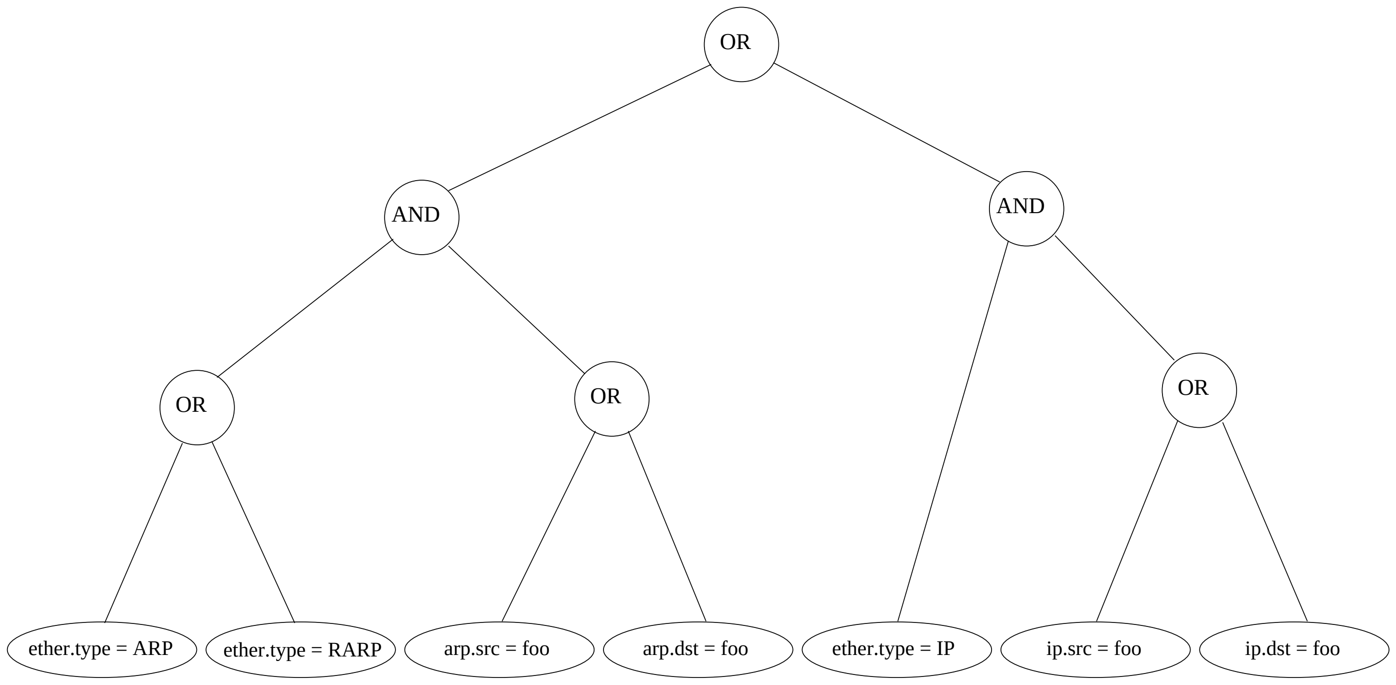

Figure 2.2: Tree Filter for host “foo”

The Berkeley Packet Filter(BPF)

BPF 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 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. 2.3.

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 2.3) is represented in CFG model as shown in figure 2.4.

However, BPF also does not provide support for filter composition. So BPF does not scale well when there are a large number of filters.

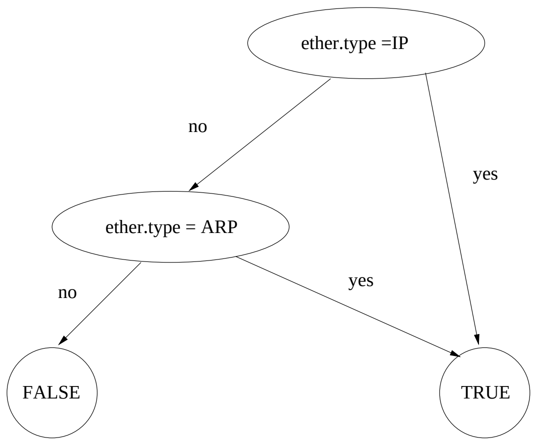

Figure 2.3: CFG Model

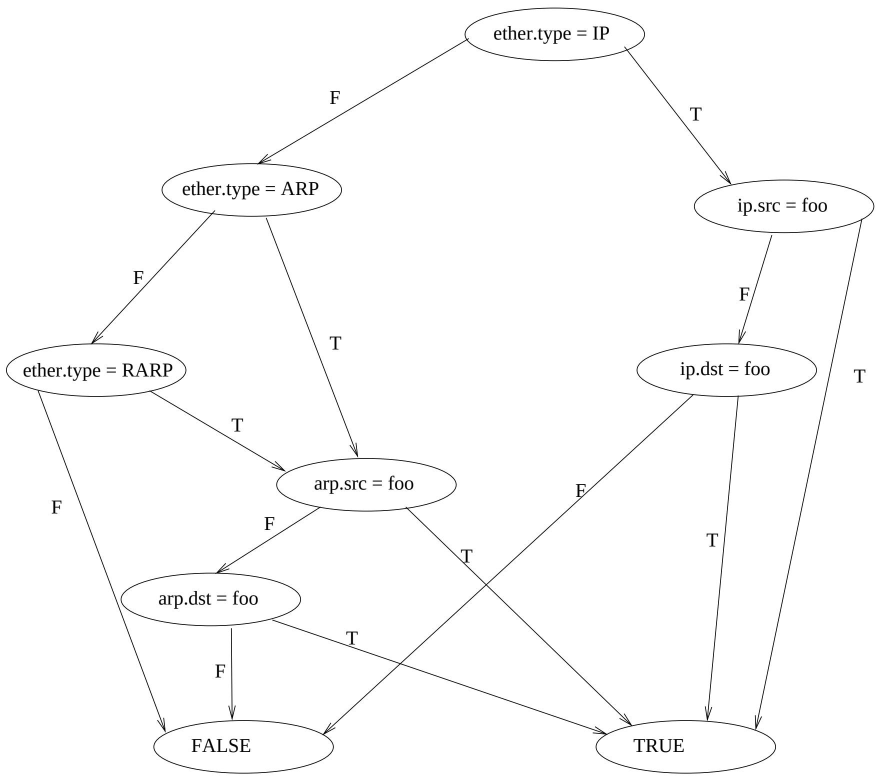

Figure 2.4: CFG Filter for host "foo"

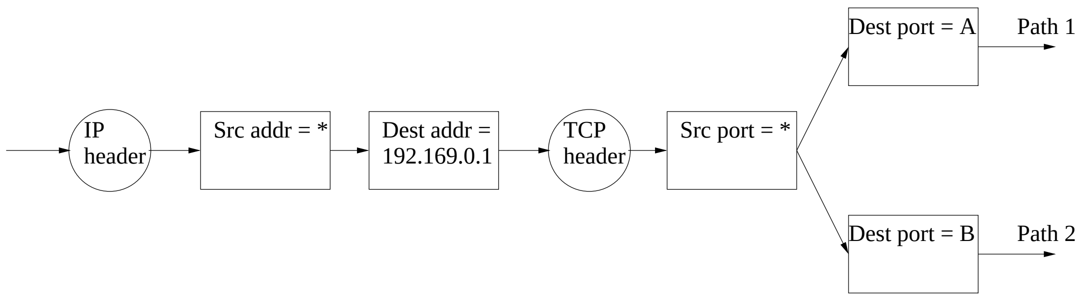

Figure 2.5: Composite filters in PATHFINDER

The Mach Packet Filter(MPF)

The Mach packet filter 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 composition of different filters.

PATHFINDER

PATHFINDER 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 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 2.5.

As these optimizations only consider common prefix, they fail when the predicates are reordered.

Dynamic Packet Filter(DPF)

DPF uses an approach of template-matching similar to PATHFINDER. DPF coalesces longest common prefix and performs some additional local optimizations to eliminate unnecessary computation. DPF uses dynamic code generation to achieve performance improvements over other interpreter-based systems.

Common prefixes always appear in the same order because DPF enforces in-order packet header traversal. But, prefix compression fails when the filter itself does not conform to same order as other already installed filters.

BPF+

BPF+ 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 data-flow algorithm, called "redundant predicate elimination" for predicate optimization. BPF+ uses other common compiler optimizations like peephole 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 2.6. Here, MPF, PATHFINDER and DPF 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. This reduces the opportunities for sharing common tests.

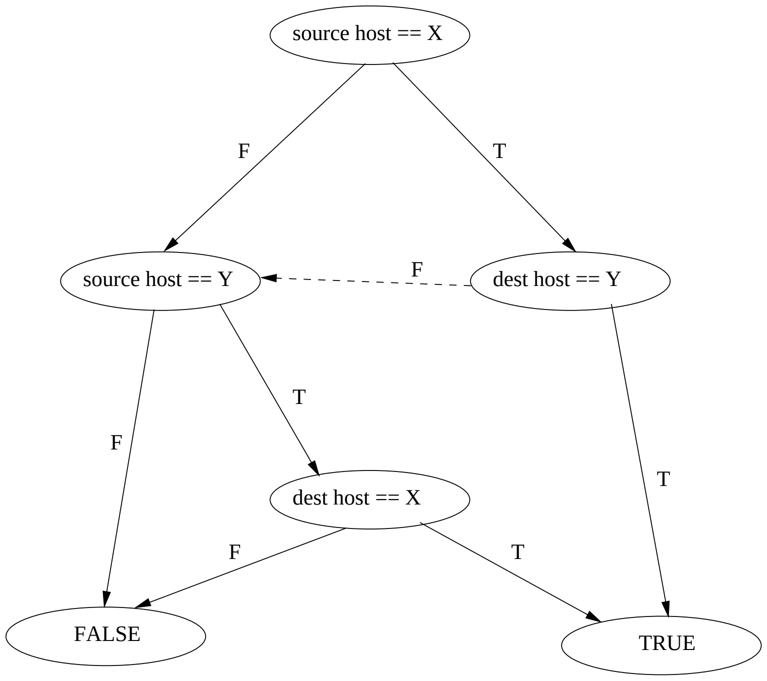

Figure 2.6: CFG for “all packets sent between X and Y”

Chapter 3

Algorithm

In this chapter we discuss the algorithm that we developed for generating packet filters. We begin this chapter with an overview of the Adaptive Pattern Matching technique.

3.1 Adaptive Pattern Matching

We have seen in chapter 1 that pattern matching techniques typically preprocess the patterns into a DFA-like automaton that can rapidly select the patterns that match the input term. This automaton is typically based on left-to-right traversal of the patterns. The main advantage of such a matching automaton is that it does not involve backtracking. The drawback of this automaton is that the size of the automaton might be exponential in the number of patterns. One way of minimizing both space and matching time requirements is to modify the traversal order to suit the set of patterns.

Consider a network packet with fields $f1$, $f2$, $f3$, ... fn. Suppose we have patterns $p1$, $p2$, and $p3$ that specify constraints on fields $f1$, $f2$, $f3$, and $f4$ as follows:

$p1 : (f1 == a)\&\&(f3 == a)\&\&(f4 == b)$

$p2 : (f1 == a)\&\&(f2 == b)\&\&(f3 == a)\&\&(f4 == a)$

$p3 : (f1 == a)\&\&(f3 == a)\&\&(f4 \neq a)\&\&(f4 \neq b)$

Then a automaton based on left-to right traversal is shown in figure 3.1.

Here, each state corresponds to the prefix of the packet seen in reaching that state and is annotated with the set of patterns that can possibly match. For example, in the figure 3.1 state 5 is annotated with $\{p1, p2, p3\}$ because after seeing the prefix aba in the packet we can not rule out a match for any of the three patterns.

Now if we consider a different traversal order, say as shown in figure 3.2 then we get an automaton that is smaller and takes less time to match patterns $p1$ and $p3$.

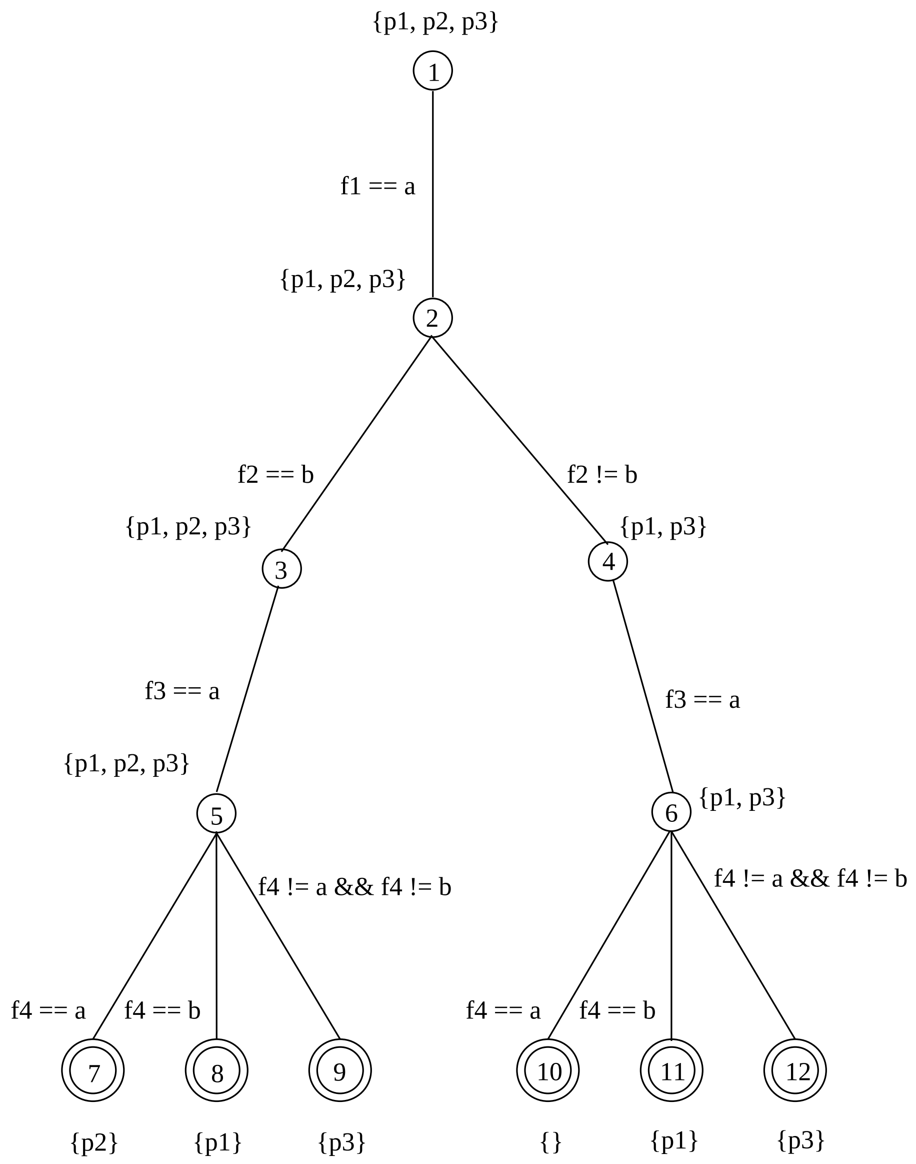

Figure 3.1: Left-to-right Automaton

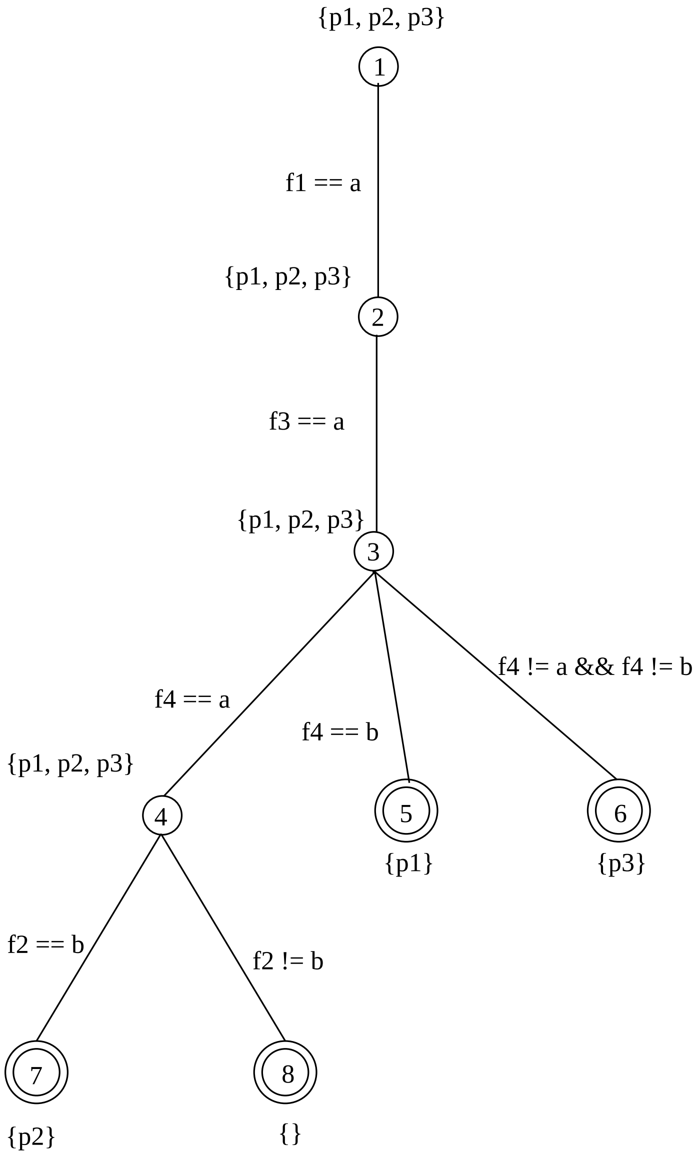

Figure 3.2: Adaptive Automaton

```
procedure Build($s$) {
1.  // $s$ is a state of the automaton.
2.  if ($s.Candidates == \phi$) {
3.      //stop
4.  } else {
5.      $tests$ = Select($s.Conditions$)
6.      //This selects the tests to be preformed .
7.      for each $t_i \in tests$ do
8.          $s_i$ = Create()
9.          for each $c_j$ in $s.Conditions$ do
10.             if ($c_j/t_i == TRUE$) {
11.                 $s_i.Matched = s_i.Matched \cup \{p_j\}$
12.             } else if ($c_j/t_i == FALSE$) {
13.                 $s_i.Failed = s_i.Failed \cup \{p_j\}$
14.             } else {
15.                 $s_i.Candidates = s_i.Candidates \cup p_j$
15.                 $s_i.Conditions = s_i.Conditions \cup \{c_j/t_i\}$
16.             }
17.         enddo
18.         Build($s_i$)
19.     enddo
20. }
```

Figure 3.3: Algorithm for constructing adaptive automaton

3.2 Algorithm to build Adaptive Automaton

Figure 3.2 shows our algorithm *Build* for constructing an adaptive automaton. A state s of the automaton remembers the patterns that are matched in reaching s from the start state using the set $Matched$. The set $Failed$ in state s contains the patterns that can not be matched after s is reached. The set $Candidates$ contains the patterns that can possibly match after s. In the start state $Matched$ is empty and all patterns are in $Candidates$. $Conditions$ is the set containing all the conditions that remain to be checked once the state s is reached. All elements of $Conditions$ are conjunctions of $tests$. $Select$ is a procedure that returns the next $tests$ that should be performed. Note that all the tests returned by $Select$ contain a common

T1	T2	T1/T2	Additional Conditions
T	T	TRUE	
T	!T	FALSE	
x == a	x == a'	FALSE	a $\neq$ a'
x == a	x $\in$ [a1, a2]	x == a FALSE	a $\in$ [a1, a2] otherwise
x == a	T	x == a	
x $\neq$ a	x == a'	TRUE	a $\neq$ a'
x $\neq$ a	x $\in$ [a1, a2]	TRUE	(a < a1) || (a > a2)
x $\in$ [a1, a2]	x == a	TRUE FALSE	a $\in$ [a1, a2] otherwise
x $\in$ [a1, a2]	x $\in$ [a3, a4]	TRUE FALSE	(a1 $\leq$ a3) && (a2 $\geq$ a3) (a2 < a3) || (a1 > a4)
T	x == a	T[x$\leftarrow$a]	
T	T'	T	

Table 3.1: Definition of / and % operations

attribute. Different patterns might contain different *tests* on the same *attribute*. For instance, a pattern may specify the *test* `p.protocol == IP_TCP` while another pattern may specify the *test* `p.protocol == IP_UDP`. There are transitions from s for each distinct *test* t_i to a new state s_i. There will also be a transition from s on *else* which will be taken when other edges leaving s can not be taken. So *else* corresponds to $!t_1 \wedge !t_2 \wedge ...!t_n$. The *else* transition leads to state s_{n+1}.

For each new state s_i we have to form the sets $Matched_i$, $Failed_i$, $Candidates_i$, and $Conditions_i$. To this end, we define two new operations, *quotient* (/) and *remainder* (%), on tests and conjunctions. Informally, if t_1 and t_2 are tests, then t_1/t_2 gives the value of the test t_1 given that t_2 evaluates to true. $t_1\%t_2$ is equal to t_2 if t_1/t_2 equals TRUE and is UNDEFINED otherwise. % is used to extend / for conjunction of tests as follows:

$(t_1 \;\&\&\; c) \;/\; t = (t_1 \;/\; t) \;\&\&\; (c \;/\; (t_1 \;\%\; t))$

where c is a conjunction and t is a test.

Table 3.1 defines the / operation. The entries are tried top to bottom, and the first applicable entry is used. Note that the additional condition in the table 3.1 must be satisfied before that entry can be used.

Now the sets $Matched_i$, $Failed_i$, $Candidates_i$, and $Conditions_i$ can be formed as shown in the figure 3.2.

1. Convert conditions to Disjunctive Normal Form.

   As we have seen in section 2.2.5, a pattern consists of an event and possibly the associated condition. Each condition is a boolean expression involving tests on attributes. So initially we have a set of conditions and their associated patterns. Let us denote this set by

$$\{[C_1 \rightarrow P_1], [C_2 \rightarrow P_2], ..., [C_n \rightarrow P_n]\}$$

There may also be patterns which do not have any associated conditions. We collect these patterns into the $Matched$ set i.e

$$Matched = \{M_1, M_2, ..., M_k\}$$

Here each M_i is a pattern with no associated condition. We collect the patterns which have conditions into the $Candidates$ set i.e

$$Candidates = \{P_1, P_2, ..., P_n\}$$

Note that initially $Failed = \phi$.

Now all the conditions are converted into Disjunctive Normal Form. We collect all conjunctions obtained from all conditions into the *Conditions* set. Note that each conjunction is associated with a pattern. This step may break up the condition associated with a pattern into multiple conjunctions. So we get the set

$$Conditions = \{[c_1 \rightarrow p_1], [c_2 \rightarrow p_2], ..., [c_m \rightarrow p_m]\}$$

where each c_i is a conjunction and $p_i \in \{P_1, P_2, ..., P_n\}$. Here, each c_i is a conjunction of *tests*. (We can consider each conjunction as a list of tests). Each *test* is either of the form (e *relop* e') or !(e *relop* e') where e and e' are expressions and *relop* is a relational operator.

$$relop \in \{=, \neq, >, <, \geq, \leq\}$$

The following example illustrates this step. Consider the following three patterns before this step:

```
P1: rx(p) | (p.protocol != IP_TCP) && (p.protocol != IP_UDP) &&
            (p.protocol != IP_ICMP) && (p.protocol != IP_IGMP)
P2: rx(p) | (((p.udp_sport == ECHO_PORT) &&
              (p.udp_dport == CHARGEN_PORT)) ||
             ((p.udp_dport == ECHO_PORT) &&
              (p.udp_sport == CHARGEN_PORT)))
```

```
P3: rx(p) | (p.tcp_flag == SYN) || (p.tcp_flag == ACK)
```

After converting the conditions into DNF form we get,

```
c1: (p.protocol != IP_TCP) && (p.protocol != IP_UDP) &&
    (p.protocol != IP_ICMP) && (p.protocol != IP_IGMP)
c2: ((p.udp_sport == ECHO_PORT) && (p.udp_dport == CHARGEN_PORT))
c3: ((p.udp_dport == ECHO_PORT) && (p.udp_sport == CHARGEN_PORT))
c4: (p.tcp_flag == SYN)
c5: (p.tcp_flag == ACK)
```

The sets constructed for the start state are $Matched = \phi$, $Failed = \phi$, $Candidates = \{P_1, P_2, P_3\}$, $Conditions = \{[c_1 \rightarrow P_1], [c_2 \rightarrow P_2], [c_3 \rightarrow P_2], [c_4 \rightarrow P_3], [c_4 \rightarrow P_3]\}$

2. Normalize variable names

   BMSL supports variables which have different scopes. *State variables* have global scope while the scope of *local variables* is restricted to a single rule. The event arguments are also treated as variables.

   This step is needed to ensure that identical variables denote identical things across all conjunctions. For instance, $\$i$ should mean the i^{th} argument of current event regardless of which pattern the conjunction belongs to. In case of state variables, identically named variables across different conjunctions denote the same thing.

3. Introduce temporary variables

   We replace expressions involving arithmetic operations with temporary variables. This allows us to compute common subexpressions only once. We ensure that identical expressions across all conjunctions get replaced by identical temporary variables. For example, let $v1$, $v2$, etc. represent variables and f be a predicate on the variables and c_j be conjunctions:

   $c_1 : !((v1 + v2) > v3 - (v4 + v5)) \ \&\& \ (2 \neq (v1 + 5))$

   $c_2 : \ ((v4 + v5) + (v6 + v7) < (v1 + 5)) \ \&\& \ !(v7 == 2) \ \&\& \ !f(v1, v2 + v3)$

   $Conditions = \{[c_1 \rightarrow P_1], [c_2 \rightarrow P_2]\}$

   So we replace the expressions by temporary variables $temp_k$ as follows:

   $temp1 = v1 + v2$

$temp2 = v4 + v5$

$temp3 = v1 + 5$

$temp4 = v3 - temp2$

$temp5 = v6 + v7$

$temp6 = temp2 + temp5$

$temp7 = v2 + v3$

$temp8 = f(v1, temp7$

So the conjunctions are now transformed to:

$c_1 = \{!(temp1 > temp4), (2 \neq temp3)\}$

$c_2 = \{(temp6 < temp3), !(v7 == 2), !temp8\}$

We note here that the expression associated with a temporary variable is evaluated only once.

4. Introduce new bit-masking expressions

   Expressions involving bit-mask operations are treated in the following way:

   (a) When we come across an expression of the form $(x\&a == e)$, where x is a variable, a is a constant and e is an expression, we look for another bit-masking operation on the same variable with a different mask $(x\&a' == e')$.

   (b) We break this operation into multiple pieces such that they mask out disjoint bits of x. For example, there could be three such masks in this case, $a1 = a\&a'$, $a2 = a\& \ a'$, and $a3 = \ a\&a'$.

   (c) Replace $(x\&a == e)$ with $((x\&a1 == e\&a1)$ && $(x\&a2 == e\&a2))$ and $(x\&a' == e')$ with $((x\&a1 == e'\&a1)$ && $(x\&a3 == e'\&a3))$.

   (d) Go back to (a) until no pair satisfying (a) is found.

   Not that the transformations are applicable regardless of which side of '==' symbol the masking operation appears.

   The purpose of the above transformations is that tests of the form $(x\&0x7f == 0x33)$ and $(x\&0x83 == 0x3)$ can be *factorized* appropriately. This is best illustrated by an example.

   Consider the $Conditions$ containing

   $[(x\&0x7f == 0x33) \rightarrow P_1]$

   $[(x\&0x83 == 0x3) \rightarrow P_2]$

$[(x == 0x20) \rightarrow P_3]$

If temporary variables are introduced without the above transformations, then we get:

$temp1 = x\&0x7f$

$temp2 = x\&0x83$

$[(temp1 == 0x33) \rightarrow P_1]$

$[(temp2 == 0x3) \rightarrow P_2]$

$[(x == 0x20) \rightarrow P_3]$

This suggests that the three tests are independent. However, with the above transformations we get:

$temp1 = x\&0x3$

$temp2 = x\&0x7c$

$temp3 = x\&0x80$

$[((temp1 == 0x3)\&\&(temp2 == 0x30)) \rightarrow P_1]$

$[((temp1 == 0x1)\&\&(temp3 == 0x0)) \rightarrow P_2]$

$[((temp1 == 0x0)\&\&(temp2 == 0x20)\&\&(temp3 == 0x0)) \rightarrow P_3]$

In this case, tests on a single attribute i.e. *temp*1 will distinguish between the three patterns.

5. Convert inequalities involving a constant into a range

   This step is used for integer variables. The purpose of this step is to combine the tests involving the same variable. The expressions that involve the relational operators $\{=, >, <, \geq, \leq\}$ and where one argument is an integer variable and the other is a constant, are converted to ranges. For instance, $x \geq 5$ get converted into $x \in [5, \infty]$, $y < 3$ into $y \in [0, 2]$ if y is an unsigned integer and into $y \in [-\infty, 2]$ if it is a signed integer.

   This step helps us merge tests and remove conjunctions containing *incompatible* tests. For instance, if a conjunction contains tests $(x > 9)$ and $(x < 21)$, then they are merged to give $x \in [10, 20]$. If on the other hand the tests were $(x < 9)$ and $(x > 21)$ then the conjunction would be removed from *Conditions* since x can not satisfy both the tests at the same time.

   Note that tests like $(x > y)$ that do not contain any constant and $(x \neq 30)$ that involve $\neq$ are not transformed.

6. Push NOTs inside and convert boolean tests into expressions

   In this step, we push ! operator inside the expressions. So, a test like $!(x > y)$ gets turned into $(x \leq y)$.

   Also, boolean tests like $(temp1)$ and $!(temp2)$ are converted to $(temp1 == TRUE)$ and $(temp2 == FALSE)$.

   All tests involving only constants are evaluated. If any of these tests like $2 > 5$ evaluates to $FALSE$ then the conjunction is removed from $Conditions$. This step evaluates the expressions involving only constants in temporary variable bindings also. For instance, if $(temp1 = 2 + 5)$ then we can replace $temp1$ by 7 in all tests.

   At this point, each test is of the form (var relop var) or (var relop const), and each temporary variable binding is of the form temp = op(e_1, ..., e_n) where each e_i is a constant or a variable, and op is an arithmetic or a bit-masking operation.

   For example the conjunctions at the end of step 3 are converted to:

   $c_1 = \{(temp1 \leq temp4), (2 \neq temp3)\}$

   $c_2 = \{(temp6 < temp3), (v7 \neq 2), (temp8 == FALSE)\}$

   The temporary variable bindings remain the same.

7. Order $Conditions$

   This step is an ordering operation on the $Conditions$ set that allows us to perform many operations efficiently in the automaton construction. For example, when the tests within a conjunction are in a particular order, we can perform certain *short circuit* operations like ignoring all tests after a particular test while scanning.

   The ordering is done in a bottom-up fashion as follows.

   (a) Order variables within each test

      If a test contains a constant and a variable, then the constant goes on the right hand side. If the test contains two variables, then the variable that is earlier in the lexical order goes on the left hand side. For example, the conjunctions at the end of step 6 are converted to:

      $c_1 = \{(temp1 \leq temp4), (temp3 \neq 2)\}$

      $c_2 = \{(temp3 > temp6), (v7 \neq 2), (temp8 == FALSE)\}$

   (b) Order tests within each conjunction

The tests in a conjunction are ordered such that a test $(x\ op\ e)$ appears before $(x'\ op'\ e')$ whenever $(x, op, e) < (x', op', e')$. In this context, a constant is greater than any variable. For operators, we use lexical ordering just as is done for variables. For example, the second and third tests are exchanged in c_2 as:

$c_1 = \{(temp1 \leq temp4), (temp3 \neq 2)\}$

$c_2 = \{(temp3 > temp6), (temp8 == FALSE), (v7 \neq 2)\}$

(c) Order conjunctions in *Conditions* We sort the conjunctions such that the smallest conjunctions (in terms of the number of tests in them) appears first. In our example no reordering is necessary as $|c_1| < |c_2|$

3.3 Computation of <determinism, utility, branching factor>

The *Select* function scans through the tests in the conjunctions until a *candidate test*, say T is identified.test For this test, *Select* computes the value <determinism, utility, branching factor>. If the best possible value is obtained then the search is stopped. Otherwise, the search continues with other candidate tests. The test with the best value i.e. highest determinism, highest utility, and lowest branching factor is returned by the *Select* function.

Next we see how the value <determinism, utility, branching factor> is computed. Depending on the type of T there are three cases as follows:

1. T is of the form $(x = a_1)$

   We identify all conjunctions that have a test of the form $(x = a_i)$ where $a_i \neq a_j \forall i, j$. All these conjunctions contribute 1 to determinism and utility. Let the distinct constants compared with x be $a_1, a_2, ..., a_k$. Then branching factor is equal to k. This is because if this test is chosen, then branches will formed for each of

   $T_1 : (x = a_1), T_2 : (x = a_2), ..., T_k : (x = a_k)$

   The else branch corresponds to performing the test,

   $T_{k+1} : (x \neq a_1) \wedge (x \neq a_2) \wedge ... \wedge (x \neq a_k)$

   A conjunction c' that does not contain a test of the form $(x = a_i)$ contributes $1 - \frac{m-1}{m}$ to determinism, where m is defined as follows.

   m is the number of different T_i's such that c'/T_i is not $FALSE$.

   c' contributes 1 to utility if it contains a test of the form x ∈ [a, a'] or x = y.

2. T is of the form x ∈ [a1, a2]

   We identify all tests in all conjunctions that are of the form x ∈ [a1', a2']. We pick a value a and compute the following values,

   n_a^+ = number of conjunctions with a test of the form (x ∈ [a1', a2']) such that a1' > a

   n_a^- = number of conjunctions with a test of the form (x ∈ [a1', a2']) such that a2' ≤ a

   n_a^* = number of conjunctions with a test of the form (x ∈ [a1', a2']) such that a1' ≤ a and a2' > a

   We compute $cost_a$ as $(n_a^+ + n_a^*)^2 + (n_a^- + n_a^*)^2$

   We select a value of a that minimizes $cost_a$. This corresponds to performing a test (x > a) at the current state of the automaton. For this test, the value of determinism is given as follows. Each conjunction c in *Conditions* contributes 1 to determinism if either $c/T = FALSE$ or $c/!T = FALSE$. Each conjunction with a test of the form (x ∈ [a1, a2]) contributes 1 to utility.

   Note that the minimum possible cost is $\frac{n^2}{2}$, where n is the number of conjunctions that have a test of the form $(x \in [a_i, a_j])$. This is minimum cost is achieved when $n_a^* = 0$ and $n_a^+ = n_a^- = \frac{n}{2}$. This means that when $cost_a$ is minimum, the intervals are partitioned into two sets.

   The branching factor is 2 corresponding to tests $(x > a)$ and $(x \leq a)$.

3. T is of any other form

   The branching factor is 2 corresponding to tests T and !T. Each conjunction c in *Conditions* such that $c/T = FALSE$ or $c/!T = FALSE$ contributes 1 to determinism. Every c such that $(c/T \neq c)$ and $(c/!T \neq c)$ contributes 1 to utility.

3.4 Subautomaton sharing

We further minimize the space requirements by merging equivalent states. Two states are equivalent if they have the same remaining conjunctions i.e the same *Conditions* sets. This merging is done at the time of automaton construction itself. When we recognize that a new state that is to be created has an equivalent state that is already created, then we make a transition to the equivalent state. Note that we don't have to create the new state. Also, we do not perform *build* on the equivalent state because its subautomaton has already been *built*. This creation of directed acyclic graph(DAG) automaton further reduces the size of the automaton. Note that this subautomaton sharing does not help in improving the matching time.

Chapter 4

Summary

4.1 Implementation

Our implementation consists of a compiler and a runtime system. The compiler is responsible for translating the network packet filter specifications into C++ code. The aspects of compilation unique to our system include type-checking for packet data types and the compilation of pattern-matching. The C++ code generated by our compiler is compiled by a C++ compiler and linked with the runtime system to produce the network packet filtering system.

Type-checking for Packet types: We have seen in section 2.2.4 that structures are augmented with constraints. These constraints have to be checked before any field of the structure is accessed. So in the type-checking phase, we add the struct-constraints to the structure fields in the tests. For example, when the type-checker comes across a test of the form `p.protocol != IP_TCP`, it resolves `p.protocol` to the `protocol` field in `ip_hdr`. Since `ip_hdr` derives from `ether_hdr` with the struct-constraint `e_type == ETHER_IP`, the type-checker associates the struct-constraint `p.e_type == ETHER_IP` with `p.protocol`.

In the type checking phase, whenever we come across any structure field in a test, we add the struct-constraint associated with the field as a test into the conjunction. Note that the struct-constraint may itself involve test on a field of another structure. Then we also have to add the struct-constraint associated with this field to the conjunction. So in this step we check each test in a conjunction and add the struct-constraints associated with any field access. Then we apply this step recursively to the struct-constraints that are added.

Let us consider the the patterns specified in the previous step. In conjunction c_1, before we can access the `protocol` field of the `ip_hdr` we need to check the struct-constraint `e_type == ETHER_IP`. So we add this struct-constraint to c_1. The conjunction c_1 becomes

```
c1: (p.e_type == ETHER_IP) && (p.protocol != IP_TCP) &&
    (p.protocol != IP_UDP) && (p.protocol != IP_ICMP) &&
    (p.protocol != IP_IGMP)
```

Similarly, c_2 and c_3 become

```
c2: (p.e_type == ETHER_IP) && (p.protocol == IP_UDP) &&
    ((p.udp_sport == ECHO_PORT) && (p.udp_dport == CHARGEN_PORT))

c2: (p.e_type == ETHER_IP) && (p.protocol == IP_UDP) &&
    ((p.udp_dport == ECHO_PORT) && (p.udp_sport == CHARGEN_PORT))
```

The above conjunctions show many fine points. First, notice that even though both `udp_sport` and `udp_dport` have struct-constraint `protocol == IP_UDP` we add it only once in each conjunction. This sharing of struct-constraints in a conjunction reduces redundancy. This is because we need to check `protocol == IP_UDP` only once along a path of the automaton. Notice that the sharing does not take place across conjunctions. This is to ensure that the struct-constraint is checked in the paths for each each of the conjunctions. Second, `protocol` itself has a struct-constraint `e_type == ETHER_IP` which is added to the conjunction.

Compilation of Pattern-Matching: We have seen in chapter 3 how the patterns are compiled into a kind of automaton for efficient pattern matching.

We used some sample pattern files to check the size of the generated automaton. We observed that a filter specification consisting of 27 patterns and 106 constraints gets compiled into an automaton having 240 states. We compiled another specification consisting of 22 patterns and 73 constraints. The generated automaton had 167 states. This shows that the size of the automaton is not exponential in the number of patterns.

Runtime System: The runtime provides support for capturing network packets either from a network interface or from a file. The code for doing this is currently based on the Berkeley packet filter code. This code is used to read all network packets (either from a file or a network interface). The actual filtering and other processing is done by the code generated by our compiler.

4.2 Conclusion

We believe that our algorithm for fast pattern matching in packet filtering can be used in many applications like intrusion detection systems, firewalls, routers and networking monitoring systems.

Bibliography

[1] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. Pathfinder: A pattern-based packet classifier. In *Operating Systems Design and Implementation*, pages 115–123, 1994.

[2] Andrew Begel, Steven McCanne, and Susan L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *SIGCOMM*, pages 123–134, 1999.

[3] S. Chandra and P. McCann. Packet types, 1999.

[4] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM*, pages 53–59, 1996.

[5] L. Hyafil and R. L. Rivest. Constructing optimal binary decision trees is np-complete. In *Information Processing Letters*, pages 5(1):15–17, 1976.

[6] M. Jayaram, R. Cytron, D. Schmidt, and G. Varghese. Efficient demultiplexing of network packets by automatic parsing, 1994.

[7] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, pages 259–270, 1993.

[8] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, volume 21, pages 39–51, 1987.

[9] R. Sekar, Y. Guang, S. Verma, and T. Shanbhag. A high-performance network intrusion detection system. In *ACM Conference on Computer and Communications Security*, pages 8–17, 1999.

[10] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. In *Automata, Languages and Programming*, pages 247–260, 1992.

[11] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *USENIX Winter*, pages 153–165, 1994.