Hardware-constrained Packet Classification

David Antoš

A thesis submitted for the degree of
Doctor of Philosophy at
The Faculty of Informatics, Masaryk University

Brno, Czech Republic
February 2006

Except where otherwise indicated, this thesis is my own original work.

David Antoš

Brno, Czech Republic
February 2006

Acknowledgements

I am very grateful to my adviser prof. Luděk Matyska for his support and guidance throughout my work, his patience and insightful comments.

My special thanks go to dr. Petr Holub and Vojtěch Řehák who helped me greatly with preparation of this thesis and spent hours in numerous discussions with me. Petr was usually the first reader of my first drafts, reworked versions of first drafts, reworked versions of reworked versions, suffering all the pain this work causes.

I appreciate work of my bachelor and master students, Kateřina Minaříková, Marek Pospíšil, Josef Ševčík, Ondřej Chaloupka, and my colleagues of the Liberouter project.

My thanks extend to dr. Eva Hladká, head of the Laboratory of Advanced Network Technologies of the Faculty of Informatics. Her support and comments to the text have been invaluable. I have had the honour and pleasure to share the laboratory with great people, Miloš Liška who also proofread the text, Lukáš Hejtmánek, Jiří Denemark, Tomáš Rebok, and others. Thanks for their support and friendship.

D. A.

Typeset by ConTEXt `http://www.pragma-ade.com`

Abstract

The goal of this work is to propose a unified packet classification method combining routing, level 3-to-level 2 address translation (ARP), and packet filtering, that can be implemented on a single packet classification unit with a content addressable memory (CAM) and a static RAM. As the target architecture of this work is a general purpose computer equipped with a hardware acceleration card and the operating system being able to classify all the packets even though at lower rate compared to the acceleration card, the packet classification may be partially postponed for the operating system. The ultimate goal is to maximise the amount of traffic confined to the hardware classification unit.

The method of combining routing, ARP, and packet filtering into a single look-up operation consists of two separate parts. First, routing and ARP are combined to a single trie-based structure called routing-ARP table that solves both steps at once. We have introduced a formalism for routing and ARP and we have proven correctness of the combined structure.

Second, the novel representation of packet filters called Filtering Decision Diagram (FDD) is added. The FDD is a special type of binary decision diagrams with semantical relationship among nodes. Compared to previous decision diagram representations of packet filters, FDDs reflect natural properties of filters and incorporate the concept of encoding rule order directly to the structure. Granularity of a decision node corresponds to a single term of a filtering language, as opposed to binary decision diagram representations where a single bit is the unit of processing (which is not feasible in CAMs). We describe procedures to build FDD representations out of packet filters.

Based on the principle of distributing relevant filters to relevant portions of destination address space, the routing-ARP table is combined with FDD representations of filters. We define procedures to rewrite the resulting structure into the target hardware architecture, i.e., first match CAM and comparison instructions that finish the classification. Rewriting the resulting structure into the target architecture gives also a general procedure to convert decision diagrams into first match CAMs.

Properties of the target hardware architecture limit the degree of freedom in the method. The method is optimised for memory conservation, adaptable to various target memory sizes and insufficient resources of an actual hardware design.

Performance of the resulting structures is experimentally verified using a prototype implementation simulation.

Contents

List of Figures

List of Tables

List of Definitions and Theorems

List of Algorithms

List of Abbreviations

ARP	Address Resolution Protocol
ATM	Asynchronous Transfer Mode
BDD	Binary Decision Diagram
BPF	Berkeley Packet Filter
BSD	Berkeley Software Distribution
CAM	Content Addressable Memory
CIDR	Classless Inter-Domain Routing
CRC	Cyclic Redundancy Checksum
CVS	Concurrent Versions System
DFS	Depth-First Search
DRAM	Dynamic RAM
ESP	Encapsulation Security Protocol
FDD	Filtering Decision Diagram
FIS	Fat Inverted Segment (Tree)
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HFE	Header Field Extractor
HSL	Hic sunt leones
ICMP	Internet Control Message Protocol
ICMPv6	Internet Control Message Protocol version 6
IDD	Interval Decision Diagram
IP	Internet Protocol
IPFW	IPFIREWALL
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
L2	Level 2 of network architecture
L3	Level 3 of network architecture
LAN	Local Area Network
LMP	Longest Matching Prefix
LSB	Least Significant Bit
LUP	Lookup Processor
MAC	Media Access Control
MTBDD	Multi-Terminal BDD
MTU	Maximum Transfer Unit
NAT	Network Address Translation
NIC	Network Interface Card
nsim	Nanoprocessor SIMulator
OBDD	Ordered BDD
OFDD	Ordered FDD
OS	Operating System
OSI	Open Systems Interconnection (ISO 9646-1)
Patricia	Practical Algorithm To Retrieve Information Coded in Alphanumeric
PC	Personal Computer
PCI	Peripheral Component Interconnect
PF	PacketFilter
RA	Routing-ARP (Table)
RAF	Routing-ARP-Filtering (Table)
RAM	Random Access Memory
RBDD	Reduced BDD
RFC	Request For Comments
RFDD	Reduced FDD
RISC	Reduced Instruction Set Computer
RLE	Run Length Encoding
ROM	Read Only Memory
SRAM	Static RAM
TCAM	Ternary CAM

TCP	Transmission Control Protocol	VDHL	Very High Speed Integrated Circuit Hardware Description Language
TTL	Time To Live	VLAN	Virtual LAN
UDP	User Datagram Protocol		

1 Introduction

A router is for the Internet what typography is for a book: it should stay invisible, just doing its work—allowing end-points to communicate.

The Internet has become an infrastructure fully comparable with electricity, water, and gas supply, and the telephone network. The Western civilisation got dependent on the networks we compared the Internet to. The same way people got used to them, enjoying their benefits and having a serious trouble when some of them stop working, people are slowly and infallibly getting dependent on the Internet which has grown to a widely accepted medium for all sort of communication, customer relations, entertainment, work, education, and science.

To understand complex systems like infrastructure networks, we decompose them into smaller parts. The Internet is usually understood as a layered structure. In the highest level, applications are the purpose of the whole structure. Requirements for bandwidth, timing, and data loss depend on applications. The transport layer supports applications, using services of the network layer. The network layer is responsible for routing datagrams from the source to the destination host. The link layer moves packets from one node to the next node on the route. In the physical layer, a physical medium is used to transport signals. Physical links are passive, the "real work" is done by network *nodes* that support higher level services. Active nodes (as opposed to end nodes) transfer packets for others.

The performance and quality of the infrastructure depends crucially on the performance and quality of its active nodes. Unlike the transport layers, the network layer involves each host and router in the network. The performance of *routers*—the units that provide network layer services—affects directly the performance of the whole Internet. The Internet, the network of networks, relies on routers. Routers connect the networks together, being a symbol of control in the seemingly disorganised world.

While many Internet users have probably never seen a router on their own eyes (or at least they did not notice it), routers have to perform lots of functions for them: run the routing protocol in order to establish reasonable routes for their packets, switch the packets, filter the packets to protect them, and many others.

Many ways of improving performance of routers are possible. Better and more powerful hardware units can be developed and utilised in the network, better software methods can be investigated and implemented. As routers are one of the most critical components of the Internet, many researchers both from academic and commercial world work on their development.

Routers usually employ one of three types of switching fabrics [Kurose and Ross, 2001]. Switching via a bus where input ports of the router transfer packets directly over a shared bus, switching via an interconnection network which is a matrix of buses, and switching via shared memory where input and output ports work as traditional I/O devices of an operating system.

Besides the switching fabrics, another part of a router that affects critically the overall performance is the *packet classification* unit which studies packets and determines their fates on their journeys through the network.

Classification units are under extreme time pressure. Consider a 10 Gbps link and packets of 1500 B length. About 833,000 packets per second can be theoretically transferred through such link. Classification of a single packet on such link should be managed in less than 1.2 μsec, ideally, as fast as possible.

From a different point of view, routers are generally specialised computing systems. General purpose computers can be and has been used as routers, starting from a diskless PCs with the DOS operating system and the KA9Q routing software [Karn, 2005] in the late 1980's to nowadays Pentium 4 stations with a complete IP stack and a plethora of routing protocol implementations.

The architecture of PC-based routers raises questions about improving their performance. Throughput of such systems can be increased by means of a hardware accelerator that switches packets avoiding them to saturate the system bus.

Although this work is based on abstract properties of a routing system, it is necessary to present shortly the actual design of the hardware accelerated router that is the target architecture of this research.

We discuss general architecture of such system in the following Section 1.1. Problems addressed by this work originate from design of the IPv6/IPv4 hardware accelerator COMBO6. We present overview of COMBO6 architecture in Section 1.2. A reader interested in the basic assumptions without basing them on a concrete architecture may skip Section 1.2. This thesis is based on abstract properties of the router design. We summarise them in Section 1.3. The problem solved in this work is stated in Section 1.4. Section 1.5 gives a summary of thesis contributions and Section 1.6 presents an overview of the thesis and a reading plan.

1.1 Accelerated PC Routers

In this work, we are interested in a special type of router based on shared memory: a personal computer equipped with several network interface cards. Such routing platform is easily affordable and highly configurable. Off-the-shelf PC can be employed. PC routers have proven both reliability and functionality of hardware and software components comparable to middle class commercial routers [Liberouter, 2005].

Work of a router can be divided into two main parts: control and data plane. In the *control plane*, the router maintains signalling and control among nodes; in the *data plane*, data packets switching is performed. Processing power of a PC is typically satisfactory for control plane, on the opposite, data plane functionality is limited by the PC architecture.

Significant technical limitation of PC-based routers is the maximum theoretical throughput of their internal busses caused by system resources saturation. Even with a fast PCI bus (i.e., 64 bit, 66 MHz) that has theoretical throughput 4 Gbps—taking into account a packet must traverse the bus twice—maximal speed at most 2 Gbps can be achieved. Besides the bus throughput, interrupt latency limits the achievable throughput especially for small packets. The system bus is shared for both control and data plane functionality.

This bottleneck can be mitigated if data plane functionality can be at least partially off-loaded to a *hardware acceleration card*, so that substantial part of the traffic is confined to the card and avoids common buses inside the PC [Novotný et al., 2003a], [Novotný et al., 2003b]. The original software router and the router equipped with the accelerator can be seen as alternative implementations of the identical specification.

Hardware/software co-design [Micheli et al., 2002] is a development methodology that inspired development of such accelerator. Co-design refers to simultaneously designing software and custom processors the program will run on (we will see in the following Section 1.2 how this idea has been exploited in an actual design) and to splitting computation between software and hardware. Starting from a purely software PC router, we identified a part worth hardware accelerating (i.e., the data plane). Both control and data plane functionality is fully preserved in the operating system, and the operating system controls the accelerator. Moreover, if the accelerator is not capable to process a packet for any reason, it can send the packet to the operating system.

1.2 Target Hardware Architecture

Hardware accelerator COMBO6 [Novotný et al., 2002] is a PCI card based on programmable circuit—Field Programmable Gate Array (FPGA). Besides the FPGA, the card is equipped with memories and other necessary logic. The complete family of cards can be found on project web page [Liberouter, 2005].

The functionality of the FPGA is determined by its microcode (usually called *design*). The COMBO6 design is built as a sequence of small processors, *nanoprocessors*, with small instruction sets specialised for particular tasks. They represent "an extra level of indirection" in FPGA programming. Freely available compilers, debuggers, and simulators have been developed for them in the project [Höfer, 2003].

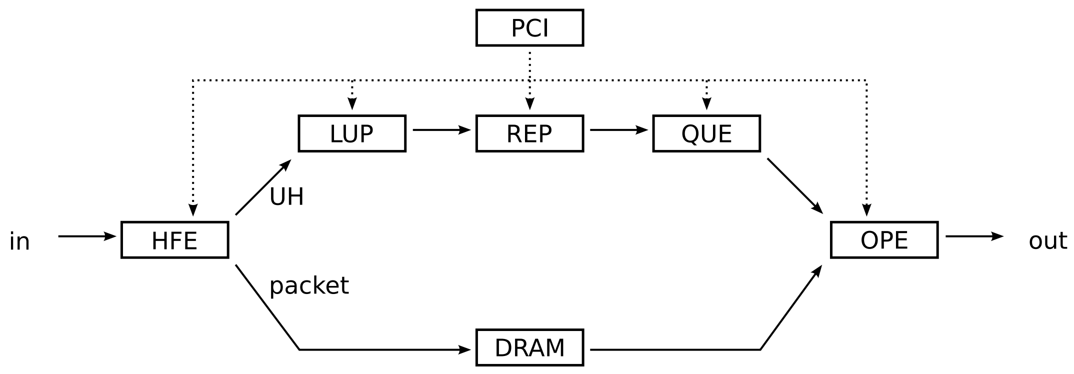


Figure 1.1 COMBO6 firmware architecture

We will describe data plane functionality of the accelerator in Section 1.2.1. Packet classification unit of the accelerator employs division of computation power between two types of memories (Section 1.2.2). Software support of the accelerator (Section 1.2.3) covers both data plane and control plane functions—the operating system handles the accelerator as a usual network interface card when transferring packets.

1.2.1 Packet Processing in Hardware

We describe in brief the data plane level of the hardware design, i.e., how a packet traverses the design (Figure 1.1).

The packet enters the system at the incoming interface and is passed to the *Header Field Extractor* (HFE) nanoprocessor. The HFE stores the packet (the payload including original headers) into the dynamic memory (DRAM). Meanwhile, the HFE parses headers of the packet and prepares a structure called *Unified Header* (UH). The UH is a fixed structure containing information from packet headers that are relevant for routing and packet filtering and "implicit" information about the packet, like incoming interface and error registers. The HFE also detects various errors in packets and marks them into the UH. The UH structure is intended to abstract from real header order and to simplify classification by using a "form," structure with fixed positions. The UH structure is described in detail in [Hažmuk, 2005], a short overview is given in Appendix A. Instruction set of the HFE nanoprocessor includes data transfers, arithmetic operations, and a rich set of bit manipulation instructions.

Lookup Processor (LUP) classifies packets on the basis of the Unified Header prepared by the HFE. The output of LUP for a packet is a control word describing where the packet is to be sent and how it will be edited before sending out. Possible LUP decisions are sending the packet to an output interface (or interfaces), sending the packet to the operating system to be processed (the software interface is understood as just a normal output interface), and/or discarding the packet. In all cases, LUP inserts the control word together with packet identification to its output queue. Details of LUP classification will be described in Section 1.2.2. LUP nanoprogram must be generated on the basis of the actual operating system knowledge about routing, ARP, and filtering.

The Packet Replicator (REP) replicates the packet identification and identification of an editing program to the Output Queues (QUE). Finally, Output Packet Editor (OPE) modifies the packet in the way prescribed by LUP and the packet is sent out. The simplest example of packet editing may be decrementing Time to Live/Next Hop Count value by one and replacing MAC addresses.

1.2.2 Unified Header and LUP

LUP is a nanoprocessor implemented in the FPGA design [Antoš et al., 2003], [Antoš and Kořenek, 2004] serving for packet classification. It uses a combination of Content Addressable Memory (CAM, see Section 2.2.1.1 for explanation how CAM works) and static random access memory (SRAM) to keep its lookup structure. We will describe the computation model of the engine. The lookup is in principle a traversal through a branching structure logically interpreted as a lookup program that is run for each incoming Unified Header. Unified Header is stored in a set of registers of constant widths.

The design of LUP addresses the problem that the width of Unified Header (which is the maximal number of bits necessary for the routing and filtering decision, about 660 bits in the actual design) exceeds significantly width of largest available CAMs (272 bits in case of the CAM employed on the COMBO6 board), especially when IPv6 is taken into account.

The instructions of LUP can be divided into three groups:

1. `CAM Step, List` is a logical representation of CAM lookup. The instruction finds a set of Unified Header registers denoted by the `List` parameter in the CAM. The registers may be selected arbitrarily. The result denotes the following instruction in SRAM to perform. This instruction can be performed at the start of the lookup program only.

2. Comparison instructions (kept in SRAM) are conditional jumps which compare UH registers with constants. On success, the jump is performed, otherwise the program continues with the following instruction in the memory.

3. Terminal instruction `EXE <status word>` returns the result. After reaching the terminal instruction, LUP inserts the status word into the output queue

and gets ready to handle another packet. The status word denotes how the packet shall be handled by replicators and output editors.

Complete instruction set can be found in [Minaříková and Höfer, 2005]. Processing in CAM and in the block interpreting the SRAM instructions is pipelined. A formal model have been developed to solve problems of memory accesses and timesharing in the hardware design [Antoš et al., 2004].

1.2.3 Software Support of the Hardware Accelerator

In the lowest level, card drivers provide basic communication with the accelerator. Moreover, the accelerator behaves as an ordinary network adapter from operating system point of view (it only switches some packets by itself "behind the scenes"). A standard network interface card device is provided by the driver.

On the control plane level, operating system kernel uses configuration of network interfaces, routing table, and packet filter setting to decide how to handle packets. These sources must be combined into the format of the classification engine LUP. Conversion of the lookup tables is provided by a user space daemon [Novotný et al., 2003b].

1.3 Properties of the Target Architecture

We have described the actual target design that motivated this work. We give an overview of abstract properties and basic features this work uses as basic assumptions:

- The accelerator can give up processing a packet and deliver it to the operating system. The operating system processes such packets as if they were obtained from an ordinary network interface card, therefore they are forwarded correctly.

- Lookup engine employs CAM as the "first stage" of the classification process. The classification is finished by interpreting comparison instructions.

- Width of the CAM is not sufficient to perform match of the full width that is necessary to classify a packet. Hence, some fields of packet headers cannot be matched in CAM and must be necessarily handled by comparison instructions.

- Parts of packet headers matched by CAM (in other words, allocation of CAM columns) are chosen globally for the whole structure.
- CAM search cannot backtrack. When a packet matches a line of CAM, it must be resolved completely by comparison instructions assigned to this line.
- CAM lookup and processing comparison instructions is pipelined. (This affects lookup time measurements.)
- CAM and comparison instructions have "distinct abilities" to express certain types of queries. CAM is suitable to perform exact match or prefix match queries while comparison instructions handle exact matches and range matches well.

We have described motivation leading to the development of hardware accelerated PC routers, and general architecture of such systems. We put the ideas into the real world describing architecture of a real system, hardware accelerator COMBO6. The design of the accelerator and the classification engine has abstract properties described above. We use them as the base for this work.

1.4 Problem Statement

The topic of this work is packet classification. The packets are classified in a hardware classification engine; we have described properties of the engine in the previous Section 1.3. The classification engine must solve all tasks related to packet classification in a router, i.e., routing, level 3-to-level 2 addressing, and packet filtering in a single lookup operation, so that a single classification unit is sufficient for the packet classification task.

The classification engine uses a CAM for a part of its decision and comparison instructions to finish the classification. We suppose that routing and level 3-to-level 2 address translation can be solved in the CAM. On the contrary, packet filtering is the entity that adds complexity, it can be solved only partly in the CAM and utilising the comparison instructions is inevitable to finish the search.

Because of hardware constraints, the classification engine is not sufficient to resolve all packets. We therefore decompose the classification problem in the following way. If the hardware classification engine is not able to classify a packet (the engine must be able to recognise its inability to do so), the packet is sent to the software to be processed.

The problem studied in this work is how to split the packet classification between hardware and software parts so that substantial part of the traffic is handled by the hardware engine without increasing the overall complexity of the system.

1.5 Contributions

The author of this work claims following contributions to the state of the art of packet classification in routers.

The central point is the proposal of a method to combine routing, level 3-to-level 2 address translation, and packet filtering into a single lookup structure suitable for a hardware lookup machine balancing computation between CAM and comparison instructions.

Following important steps had to be done in order to reach the main contribution:

- A set of previous packet classification methods has been collected and analysed. It served as a knowledge base for designing the classification engine based on partial classification in a CAM. (The proposal and design of the classification engine is a collective work, to which the author explicitly contributed.)

- A formalism to denote routing, level 3-to-level 2 address translation, and packet filtering in operating systems is introduced.

- In order to set a useful formal description of packet filtering, a representative set of real-world packet filters has been studied and their expression power is compared. Ability of the target hardware to perform packet filtering is demonstrated. Principles of transforming packet filters into the hardware classification engine are described.

- A method to combine routing and level 3-to-level 2 address translation is stated formally.

- Filtering Decision Diagrams (FDD), a Binary Decision Diagram based structure with semantic node relations for packet filter representation is introduced. Procedures to handle FDDs are defined.

- Complexity estimates of software packet forwarding and packet forwarding by the developed method are presented.

- A prototype of the method has been implemented.
- Experimental results evaluating usability of the method in terms of memory usage and time required to perform a lookup are given.

1.6 Thesis Outline

This work is intended to be read more-or-less linearly; exceptions are noted below and in introductions of sections containing parts that can be skipped for the first reading or by a reader pressed for time.

Chapter 2 gives an overview of previous methods of packet classification relevant for this work. A reader familiar with packet classification methods may skip that chapter.

Chapter 3 states formalisms to describe routing, ARP, and packet filtering. Various types of packet filters are studied and compared with respect of expression abilities of the hardware engine. In Chapter 4, a method to combine routing and ARP tables is presented. Chapter 5 introduces Filtering Decision Diagrams as a means to represent packet filters, a method to combine them with routing-ARP tables, and to rewrite them to the lookup engine using a CAM for part of the classification decision.

Experimental evaluation of the method is described in Chapter 6. Summary and conclusion is found in Chapter 7.

2 Methods of Packet Classification

No single algorithm will perform well for all cases.
– [Gupta and McKeown, 2000]

This survey is intended to guide the reader through previous methods of packet classification that inspired and are relevant for this work. The chapter is divided into two main parts, longest matching prefix lookup used mainly to evaluate routing tables in the following Section 2.1, and packet classification, i.e., classifying packets on multiple fields, in Section 2.2. Although we try to classify the methods into several groups, division lines between them are sometimes quite fuzzy.

Among longest matching prefix lookups, many methods are based on tries that represent keys as sequences of symbols (Section 2.1.1). Numerous improvements have been developed in this area, compressing paths and levels, representing tries as bitmaps for easy hardware processing and using tries for multidimensional filters. Another group of algorithms prefers choosing length of matching key first and then finding the output for the key (Section 2.1.2). A plethora of techniques tries to decrease the number of distinct prefix lengths (Section 2.1.3).

The principle of expansion/compression approach described in Section 2.1.4 indirectly inspired the naive method of combining routing with packet filters presented in Section 5.2. Our final target architecture divides the computation between two types of memories with distinct abilities. Similar problem has been studied for hierarchies of RAMs of various speeds—processor cache and system memory (Section 2.1.5).

Section 2.2 divides methods of packet classification into following main categories. Exhaustive searches (Section 2.2.1) just test all entities in the set, often employing hardware parallelism such as content addressable memories (we also use as a part of our target hardware architecture). Decomposition (Section 2.2.2) splits multiple field search problems into single fields. Tuple search (Section 2.2.3) uses the number of bits specified in each field. Decision trees (Section 2.2.4), and decision diagrams (Section 2.2.5) use parts of keys to make decisions in tree or graph structures. In Chapter 5, we develop a special type of binary decision diagram to represent packet filters in this work.

Studying the techniques helped significantly during development of the classification engine used as the target architecture in this work.

2.1 Longest Matching Prefix Lookup

This section gives an overview of existing results in the field of longest (or best) matching prefix lookup.

2.1.1 Trie and its Variants

We describe the trie data structure and its numerous improvements. Trie is one of the simplest data structures for best matching prefix lookup.

2.1.1.1 Basic Trie Structure

The trie data structure represents keys as sequences of symbols, not as a whole like conventional structures do. Let us describe now the basic version of trie as shown in [Knuth, 1998]. Let us have a finite alphabet Σ. We put $|\Sigma| = m$. Trie is an m-ary tree, its nodes are m-ary vectors indexed by the Σ alphabet. A node in depth l represents a set of keys starting with a prefix of length l. The node represents an m-way branch driven by $(l + 1)^{\text{st}}$ character of the searched word.

Trie lookup starts at the root node branching by the first character. In a general case we progress as follows. We take the next symbol of the word, let it be k. Then the field of the current node indexed by the character k keeps the pointer to the subtrie, that corresponds the lookup in the unread part of the key. Note that if the key is not in the trie, we find at least its longest prefix.

The time is linear with respect to the length of the key, the same holds for inserting and deleting. Memory complexity becomes a problem in practical applications. If we store the nodes one-by-one into a linear field, majority of the nodes is used only sparsely, wasting memory significantly.

2.1.1.2 Dynamic Packed Trie

Dynamic packed trie is an optimised version of trie. It heuristically reduces memory complexity of the structure. The price is more complicated lookup and mainly more expensive insertion.

The main idea is that the sparse nodes are kept *one mixed into another* into a linear array. A node of the trie uses fields left empty by another one. We have to distinguish the fields belonging to the node we are working with. It would be possible if we store the information about the *symbol of the alphabet* corresponding to the field. Moreover, we must not pack two nodes on the same starting position, therefore we need to add a bit denoting *node base position*.

The compression was developed by Liang [Liang, 1983], its comparison to other methods in a practical application is documented in [Antoš, 2001]. Time complexity is still linear with respect to the key length. The complexity of insertions increases depending on the implementation of the finding of a suitable position to store the node. A simple heuristics can be used—we put a node with more than certain number of used fields to the end of the array.

2.1.1.3 Patricia

Patricia (Practical Algorithm to Retrieve Information Coded in Alphanumeric) is a structure derived from the binary trie [Knuth, 1998]. The nodes with the only child are removed and each node keeps the number of bits that should be skipped before next comparison is performed. Patricia cannot find an exact match but only a *possible match*. At the end of the lookup we have to check if the result is really a match (checking the full string stored in the terminal node).

The main difference between Patricia and binary trie is that Patricia uses only bits relevant to the decision where to go. This approach is called *path compression*.

Patricia requires the stored language to be prefix-less, no string may be a prefix of another. This can be easily achieved by inserting a special end-of-word character at the end of each word.

The structure reduces the number of nodes in the tree, nevertheless the expected depth of the tree (lookup time in other words) remains the same as for binary tries. Moreover, it is not possible to extract full keys from Patricia, the compression drops parts of strings.

A variant of Patricia structure has been used to store routing tables in Berkeley UNIX operating system [Sklower, 1993].

2.1.1.4 LC-trie

The main idea of *level compressed* trie, LC-trie in short, is as follows. The highest i complete levels of the trie are replaced with a single node of length m^i and this replacement is propagated top-down. This structure was developed by Andersson and Nilsson [Andersson and Nilsson, 1993], [Andersson and Nilsson, 1994], [Andersson and Nilsson, 1995].

Before we define the LC-trie, we need several auxiliary definitions:

***i*-prefix** String v of length i is called i-prefix of string u, if a string w exists (possibly empty) so that $u = vw$. The string w is called i-suffix of the string u.

Multi-digit trie Multi-digit trie containing n elements is

- for $n = 0$ an empty leaf,
- for $n = 1$ a leaf containing that element,
- for $n > 1$ an inner node of degree 2^i for $i \geq 1$. For each possible i-prefix P it has a child that is a multi-digit trie and it contains all i-suffixes of all keys starting with the string P.

For $i = 1$ a multi-digit trie becomes an ordinary one. Previous definition does not cover all generalizations of trie, it allows only powers of two as "compression factors."

Multi-digit trie is called *dense* if it has the same amount of leaves as the corresponding binary trie.

LC-trie Level compressed trie is a multi-digit trie satisfying:

- degree of each node is 2^i, where i is the smallest number such that at least one of the node's children becomes a leaf,
- each child is an LC-trie.

LC-trie is a binary tree, where the complete levels are replaced with big nodes and this replacement is propagated top-down. Note that this compression adapts nicely to the distribution of stored words.

LC-tries have the smallest external path among all dense tries. Expected time to find a key is $\Theta(\log^* n)$ for uniform distribution of keys, where $\log^* n$ is an iterated logarithm: we put $\log^* n = 1$ and for $n > 1$ we define $\log^* n = 1 + \log^*(\lceil \log n \rceil)$.

Level compression and path compression may be combined.

Nilsson and Karlsson [Nilsson and Karlsson, 1999] used LC-trie structure for best matching prefix lookup in an implementation of a IPv4 router. They combine the basic LC-trie with path compression. To make the implementation efficient, they developed a compact representation in the only linear array (containing an exponent of node's branching factor, skip value for path compression, next node pointer/full string and output information pointer). They also give a method for tree building in $O(hn)$ time, where h is the depth of resulting LC-trie[1]. The algorithm works on previously sorted list of strings.

The problem of the basic LC-trie version is that the only missing string may cause that it is not possible to build a complete level. The solution may be that when finding the branching factor, the levels are not required to be fully complete, but we choose a weaker criterion. We choose a *fill factor* x, where $0 < x \leq 1$. When

[1] Quite a strange measure depending on the result of the algorithm.

computing the branching factor for a node covering k prefixes we use the highest possible factor that causes at most $\lceil k(1-x) \rceil$ empty leaves.

The branching factor at the root node affects the overall performance significantly, hence the paper recommends to have a fixed branching factor for the root node, independently on the value of the fill factor x.

Experimental results for IPv4 (by Nilsson and Karlsson): average depth of the trie is less than 2 and each node needs one memory access. An extra access is needed to check whether the found string is really a match (because of the path compression). If so, the next hop value is read in one more memory access. Minimally 0.5 million lookups per second can be performed in a software implementation running on Pentium 133 with fill-factor 0.5 and the first branching of the root on the 16th bit.

2.1.1.5 Tree Bitmap Algorithm

Eatherton's tree bitmap algorithm [Eatherton, 1999] is a hardware implementation of multibit trie (see Section 2.1.1.4). The basic idea is the same as with multibit tries—to compare simultaneously multiple bits of the key. The improvements in speed are reached by means of sophisticated encoding of trie nodes.

In tree bitmap algorithm, nodes of the multibit trie are coded as pairs of bitmaps. The *Internal Prefix Bitmap* identifies prefixes stored in the node in the linearized format; each row of the trie is captured top-down and from left to the right. The *Extending Prefix Bitmap* contains a bit for all possible 2^i pointers.

All the node's children are stored contiguously in the memory, allowing to have just one pointer for all children as each child node can be calculated as an offset. Finally, the last idea is to store output values for prefixes stored in the node in a separate array.

Eatherton's algorithm has been employed in the Fast Internet Protocol Lookup (FIPL) engine [Taylor et al., 2002a], [Kuhns et al., 2002]. The engine is a finite-state machine interpreting the tree bitmaps. FIPL is a part of an ATM switch called Washington University Gigabit Switch (WUGS)[2] and is implemented using Field Programmable Gate Arrays and static RAM. For IPv4, depth of the subtrees was chosen to be 4, allowing to keep tree nodes in 36-bit words. A pathological look-up requires 11 memory accesses. It allows—assuming FPGA and RAM running at 100 MHz—1,136,363 lookups per second for a single FIPL engine in the worst case.

2.1.1.6 Grid-of-Tries

Grid-of-tries [Srinivasan et al., 1998] is a trie-based memory for multi-dimensional filters, such as source-destination address pairs. The basic scheme can be ex-

[2] `http://www.arl.wustl.edu`

tended to handle filters on more dimensions, for example with port and protocol numbers.

The structure is motivated with set pruning trees. Let us show the idea behind set pruning trees first. It is a trie of tries, where we first match the destination prefix and the result is the source trie. In the source trie we match the source prefix and obtain the output information. The key is how to connect the source and the destination tries. This simple scheme suffers from memory blowup as a source prefix may appear in several tries. A worst case example uses $O(N^2)$ memory, where N is the number of prefixes in the structure.

In order to avoid memory explosion, we observe that filters associated with a destination prefix D are copied into source trie of D' whenever D is a prefix of D'. We can avoid that by pointing D to a source trie that stores output whose destination field is exactly D. This requires to modify the search strategy so that we must now search the source tries associated with all ancestors of D. Since each output is stored just once, the memory requirement is $O(NW)$, where W is the maximal length of the key. On the other hand, time cost grows to $O(W^2)$.

Search time can be improved to $O(W)$ while the memory requirement is kept linear. The key idea is using precomputation and switch pointers. The method is quite tricky, therefore we recommend the interested reader to have a look at the Section 5 of [Srinivasan et al., 1998].

Based on the observation that every packet matches at most several source-destination prefix pairs present in the rule set, Baboescu et al. [Baboescu et al., 2003] extend the grid-of-tries approach with path compression. The method is again based on precomputation that avoids backtracking.

2.1.1.7 Multi-bit Trie with Expansion

Another version of trie a bit shifted to use in hardware is presented by [Moestedt and Sjödin, 1998]. They use a multi-bit trie with number of bits in levels, typically chosen in order to break the key into three to five pieces. The prefixes are expanded up to the nearest level. There are three types of nodes: *valid*, *part*, and *invalid* ones. A valid node represents an entry in the database. A part node represents a prefix of a valid entry, and an invalid node does not represent a valid entry in the database, it has a special meaning for the lookup algorithm.

To simplify the lookup process, we want the tree to satisfy two conditions: All possible children of a part node are present in the trie (so called *prefix group*; the added prefixes that do not appear in the database are marked is invalid), and no node is allowed to be valid and part at the same time. If a prefix exists that is both valid and prefix of a valid prefix, it appears several times in the trie, once as a part node and then expanded into a prefix group where all entries are valid.

To find the result, the trie is searched from the shortest prefix up to the first valid or invalid node. The lookup time is linear wrt. the number of levels of the trie. There is a trade-off between memory and lookup time as having less trie levels causes more auxiliary entries to be added. The paper [Moestedt and Sjödin, 1998] provides measurements in a practical application.

2.1.2 Binary Search on Prefix Lengths

Binary search over prefix lengths [Waldvogel et al., 1997], [Waldvogel et al., 2001] is based on three ideas: hashing to check if the address matches a prefix of given length. Then, binary search through prefix lengths is used and finally some precomputing is done to prevent backtracking.

First, we show the linear lookup on tables containing prefixes of the same length and later we will refine this basic scheme. We divide the prefix database according to their lengths, creating a separated hash table for each length. To find the longest prefix for the key D we start from the table of the longest prefix, we extract the appropriate number of bits from D and we try to find it in the hash table. If it succeeds we have the best matching prefix. Otherwise, we continue testing tables of the nearest smaller prefix length. Having a hash function computable in linear time, this lookup is linear with respect to the number of distinct prefix lengths in the database.[3]

Another possibility is to go through the distinct prefix lengths using binary search. The problem is that we need some notion of "shorter" and "longer" in terms of prefix lengths. Alas, such kind of information cannot be obtained from hashing. Hashing answers only "hit" and/or "miss" but nothing in between.

From the other side, when we find a matching field, it does not mean that it is the best matching one. There may be a longer one. Hence, when we match, we have to continue testing longer prefixes, too. More practically, when we find a match we remember it and go on checking longer tables. We keep a track of the so far best matching candidate. On the contrary, if we miss, it only has sense to continue among shorter prefixes. To make this strategy correct, we have to add prefixes of all longer lengths to all tables of smaller lengths (if they are not already there) to lead the search. Those dummy prefixes (called *markers*) allow the algorithm above to get to longer prefixes.

If we add such an auxiliary information, natural question is how much memory we will have to pay. It turns out that markers do not have to be added to all nodes but only to a logarithmic number of them. Using a sophisticated precomputation (we again refer to the paper) and pressuming we have a constant-time hashing function, we can achieve lookup time $O(\log W_d)$, where W_d is the number of distinct prefix lengths in the database.

This basic scheme may be improved. One possibility is to move the top-speed lookup towards the most probable case (paying some performance loss in the

[3] This assumption is very important and of course quite difficult to achieve, the weakest point of the technique.

worst case). This can be done introducing some asymmetry into the structure, pushing the most probable length to the root (for IPv4 typically 24 bit length). Obviously, some paths may degenerate to a pure linear search which is unacceptable.

Another improvement may be *mutating binary search*. When we get a matching entry in the hash table and we move to a new subtree, it is sufficient to stay in this subtrie. It creates a whole network of binary trees, getting a match causes changing the tree to a more specialized one. In other words, the binary tree changes, mutates, dynamically during the search.

Each field in the hash table can contain a description of the new tree specialized for prefixes of this table. Waldwogel et al. do not show any theory for this method, they only declare that in all analysed databases the number of hash lookups decreased from five to four in the worst case and the average number of hash table lookups was two for IPv4.

This scheme can be generalised to more dimensions [Waldvogel, 2000].

2.1.3 Controlled Prefix Expansion

Controlled prefix expansion transforms a set of prefixes into an equivalent set of prefixes with fewer prefix lengths [Srinivasan and Varghese, 1999].

The prefixes are padded with all possible strings of zeroes and ones that are missing in the database to reach higher prefix length. If a string we should add with this process is already in the database, we do not add it, we leave the original one in its place. It is obvious that when performed naively, prefix expansion may significantly increase storage. For example, the whole prefix lookup problem can be converted into the ordinary array search if we expanded all the prefixes in the database to the full length. The search would be especially easy, nevertheless this solution is unacceptable as the array would require 2^{128} fields for IPv6.

On the other hand, prefix expansion can improve many lookup schemes. We want the number of distinct prefix length to be as small as possible. The paper recommends the initial lookup (16 or 24 bits for IPv4) as the start of the process. The model may be more general, picking the lengths dynamically. We will show several methods of choice of lengths in the following sections.

2.1.3.1 Choosing Optimal Levels

Using “brute force” to choose the optimal solution is not possible due to the enormous number of combinations to test. This naturally leads to dynamic programming. We will describe the method by [Srinivasan and Varghese, 1999]. Assume we wish to minimize the total number of expanded prefixes using only k lengths. In the first step, they compute the histogram of prefix lengths in the database. The last level l_k must be the total key length. In the second step, we reduce the problem to placing next-to-last level l_{k-1} and covering the remaining lengths using $k-1$

lengths. If we compute the storage required for each value, in the third step we can choose the minimal value.

The above indicated recursive algorithm reduces the optimal level picking to Wk subproblems, where W is the key length and k is the number of desired levels. Time complexity is $O(W^2k)$. The algorithm minimizes the number of expanded prefixes. The final goal is to reduce the total storage required; it depends on the actual storage method. For example, in tries some added prefixes may increase the storage significantly while others may share portions of the trie structure. Therefore an application specific solution is needed.

Let us now consider a multibit trie with prefix lengths expanded to certain levels. To search for the best matching prefix, we break the key into chunks corresponding to the lengths of the trie and use the chunks to follow the path in the trie until we reach a null pointer. Following the path, we keep track of the last output associated with the path. The last information we discovered is the longest matching prefix. The search time is $O(k)$, where k is the maximum path through the expanded trie.

When inserting, we first simulate search on the string up to the last chunk. We either terminate by reading the last chunk or reading a null pointer. In both cases we finish the chunk inserting the remaining prefix bits there.

For deletion, no way exists to find out which expanded prefixes belong to a certain prefix. The easiest way to handle deletions is to keep an auxiliary one-bit trie in the memory containing the prefixes in the pure, unmodified form. It could serve both insertions and deletions as a initial data structure from which the "productive" compressed version is derived. The complexity of insertion and deletion is $O(W)$ to search plus the time to reconstruct trie node $O(S_m)$, where S_m is the maximum size of the node.

2.1.3.2 Adaptive Level Cutting

The next natural step is to allow the expansion lengths differ in various subtries [Sahni and Kim, 2003]. Complexity of the expansion algorithm increases with a multiplicative factor of n—number of prefixes in the database, the total time complexity is therefore $O(nW^2k)$. By reformulating the dynamic programming expansion method, they achieve complexity $O(nWk)$. Nevertheless, if we compare this method with LC-tries, we discover that various paths lead to the same result. This structure is nothing else than just LC-trie. (Moreover, Sahni and Kim declare that LC-trie is a special case of their structure with the special property that all levels are completely packed. This is not completely true as there is a more sophisticated version of LC-trie allowing to set fill factor for packed nodes.)

2.1.3.3 Applying the Ideas to Other Methods

The ideas of prefix expansion are quite general and do not have to be related only to trie searches. They can also be applied to binary search on levels (see Section 2.1.2) [Sahni and Kim, 2004].

Time complexity of binary search on levels depends logarithmically on the number of distinct lengths in the database. By expansion, we can decrease the number of distinct lengths, paying with some increase of storage. Nevertheless we needed markers to make the searches work correctly. The number of markers actually decreases. When n prefixes are expanded so that only $W/2$ lengths are left in the database the number of prefixes doubles to $2n$. The worst case number of markers is then $n(\log_2 W - 2)$. It means that the total number of hashes can be reduced by one without changing the worst case storage requirement.

Compressing the levels for hash table search produces serious problem of finding a hash function. The requirement of having perfect hash function is very strong. Sahni et al. use a bit weaker criterion, semi-perfect hash function. *Semi-prefect hash function* is a hash function that guarantees that the number of collisions in a bucket is bounded from above by a constant (known in advance). This allows using fixed memory for the list of conflicting items. Obviously, this approach wastes memory as the lists may be used in a small number of cases.

The problem is that when reducing the number of levels causing the hash tables large, finding a semi-perfect hash function may be very costly operation, it can take even minutes to do.

2.1.4 Expansion/Compression Approach

Let us now have W-bit key to find the longest prefix. Expansion/compression method [Crescenzi et al., 1999a] starts from a fully expanded table with 2^W entries and tries to compress them. We can expect a limited number of different outputs among the 2^W entries in the table. Representing the relation between keys and outputs as strings, the strings can be compressed in order to provide an implicit representation of the strings in the table.

In the expansion phase, we derive implicitly the outputs for all the 2^W keys in the obvious manner (cf. Section 2.1.3, the process is similar). In the compression phase, clusters of prefixes sharing identical outputs are compressed using Running Length Encoding. A lookup in IPv4 can be performed in just three accesses (one for each half of the address, the final one to pick up the output).

2.1.5 Introducing Memory Constraints

Cheung and McCanne [Cheung and McCanne, 1999] study possibilities of dividing the best matching prefix search into a hierarchy of RAM memories having different sizes and speeds. The structure is expected to be interpreted by a general purpose processor, first type of memory is the processor cache, the other the main system memory. The method is suitable for architectures where cache movements can be explicitly requested by the native code. When cache control is not available, the method behaves as an approximation as the most often accessed entries are planned to the fast memory and they probably stay in the cache.

Suppose that we have two types of memories, with sizes and access times (S_1, T_1) and (S_2, T_2). Each prefix j has an associated probability of appearance p_j. Assuming independent packet arrival (which is not very realistic), the average prefix retrieval time is $C = \sum_j p_j(a_jT_1 + b_jT_2)$, where a_j (resp. b_j) is the number of type 1 (resp. 2) memory accesses needed to retrieve the prefix j. We moreover require that the lookup tables kept in each type of memory do not exceed the capacity of the memory. The problem now is how to structure the set of tables that minimizes average prefix retrieval time C while satisfying memory size constraints.

Cheung and McCanne use a generalized LC-trie as the base structure. By generalized LC-trie we mean an LC-trie (see Section 2.1.1.4) where all the levels are complete. This can be achieved by prefix expansion. The lookup structure is created in two stages. First, the set of prefixes is transformed to a complete prefix trie. Stage 2 transforms it to a generalized LC-trie implemented using lookup tables. The first transformation is trivial, stage 2 transformation affects the lookup performance directly and this is the focus of the optimisation.

The article [Cheung and McCanne, 1999] gives two optimisation algorithms to minimize the cost value C. The dynamic programming algorithm runs on average in $O(HnS_1^2)$ time, where H is the height of the complete binary trie (32 for IPv4, 128 for IPv6), n is the number of nodes, and S_1 is the size of type 1 memory. Although the expression looks polynomial, in fact it is exponential in the size of the input, therefore this is a pseudopolynomial algorithm. The other method is a Lagrange approximation algorithm, with average complexity $O(HnA)$, where A is the number of binary searches for Lagrange multipliers (in tests, a value around 10). The second algorithm has much better running time but it returns an approximation of the optimal solution. Both the algorithms are quite complex, therefore we do not describe them here and we refer to the paper. The result of the algorithms is a schedule how to place tables representing levels of the LC-trie into memories.

2.1.6 Other Approaches

Wang [Wang, 2005] notes that lookup performance in routers is degraded by unsuitable methods of address allocation. The IPv4 allocation schemes and small address space led to a common situation when a single entity has several non-contiguous block of address prefixes. It causes bigger routing table structures. The paper describes strategies of allocation taking address aggregation as a measure. A growth-based adaptive method is proposed, statistically evaluated and simulated. Although it is not directly related to this work, it is interesting to note that even "administrative" steps may be taken to keep routing table growth under control.

Purely hardware solutions to the problem of finding the best matching prefix can be found. Pao et al. [Pao et al., 2002] propose a hardware architecture based on binary tries partitioned into blocks of four levels that can be searched in parallel. The paper concludes that extending the scheme to IPv6 requires further research.

2.2 Packet Classification

For mere routing decision, finding the longest matching prefix of the destination address is sufficient. More general packet classification, i.e., classification on multiple packet header fields, is necessary for many network functions including firewalling, monitoring, and Quality of Service.

Entries for packet classification are called *rules*. A *rule base* or *classifier* is a finite sequence of rules. The packet classification problem is to find the first matching rule for each incoming packet at a router [Singh et al., 2003].

Although practical implementations of firewalls allow filtering on nearly all fields that can be found in packet headers (up to application level headers and even to scanning packet contents), the comparison base common in the literature is a "standard 5-tuple search," i.e., classification on source and destination addresses, source and destination port, and protocol type. It is a well chosen subset: address test is usually a (longest) matching prefix search, port numbers require range searches (the value should lie in the range specified), and protocol type is often an exact match query. From that point of view, classification schemes can be often widened to an arbitrary number of fields "by conventional means," just adding the fields to the scheme without the need to introduce additional types of searches.

The following survey is partly inspired by the classification given in [Taylor, 2004].

2.2.1 Exhaustive Search

The easiest way to solve any finite searching problem is just to test all the entities in the set. Most real-world packet filters use linear search to match their rules, i.e., the rules are just checked one-by-one until a match is found. This usually requires minimal storage and time linear to the number of rules (multiplied by complexity of testing a rule).

2.2.1.1 Content Addressable Memory

Search time may be improved introducing some level of paralellism. In the ideal case, we may match all the rules concurrently using a specialised hardware—ternary CAM (TCAM).

As opposed to standard (random access) computer memory where the user supplies an address and the memory returns the content of the memory cell addressed, a *Content Addressable Memory* (CAM) is supplied by the data and returns an address or addresses where the data is stored in the memory. The search is performed in parallel, by brute force hardware. Input keys are compared against every CAM entry simultaneously, thus performing the search in a constant number of clock cycles, independently on the position of the data in the memory. CAM designs therefore have a comparison circuit for each bit of the memory.

Binary CAM searches data containing only binary zeroes and ones. *Ternary CAM* (TCAM) supports a third state called *don't care* and often written as "X" that means that value of this bit is not taken into account during the search. The third state is usually implemented by adding a mask bit denoting whether the data bit should be checked (this doubles the storage required and adds complexity to the circuitry). Ternary CAM can be used for longest matching prefix search if the prefixes are sorted in non-increasing prefix lengths.

The price paid is in general (adapted from [Taylor, 2004])

1. high per-bit cost compared to other storage technologies,

2. high power consumption,

3. limited scalability to long keys.

2.2.1.2 Modified CAM Schemes

Spitznagel et al. [Spitznagel et al., 2003] study disadvantages of TCAM solutions, addressing mainly storage inefficiency and power consumption. They propose a scheme called Extended CAM (E-TCAM) specially adapted to the task of packet classification. Range checks are directly implemented in hardware at the price of increase of number of transistors needed. It avoids the need to replicate rules to cover a range query that cannot be easily converted to a bit mask. Power consumption is reduced by limiting the number of active regions of the device during the search. Each region has an associated index filter covering all filters in the block. A search first checks the index filters. In the second step, only the blocks with matching index filters are searched. Unfortunatelly, the device was tested on simulation only; it has never been produced physically.

Another scheme by Taylor and Spitznagel called LECAM (Label Encoded Content Addressable Memory) [Taylor and Spitznagel, 2005] decomposes the search into stages. The first stage is a collection of parallel search engines, one for each field. The engines are optimised for the type of query associated with the field (e.g., exact matching for transport protocol). The output of each engine is a set of labels corresponding to the packet headers. When the set of labels is resolved the results are fed to a modified CAM structure that returns the set of matching filters for the packet.

Ways to reduce power consumption of CAMs have been studied by Zane at al. [Zane et al., 2003] for the task of finding the longest matching prefix. CAMs are available that allow reducing power consumption by means of addressing smaller portions of the memory. The portions can be selectively included in the current search. The task is therefore to partition the CAM (containing routing table) into chunks. Given an input, right parts to search must be selected. Finally, for a given partitioning scheme, the size of the largest partition must be determined so that hardware designers can allocate a power budget. The Bit Selection architecture uses hashing to determine the chunk to search. The sets of hashing bits are selected by heuristics. Other approach, Trie Based Partitioning, uses a small index TCAM (that has to be switched always on) to select chunks of the main CAM. A post-order split algorithm is described in the paper to partition the main CAM contents. Lu [Lu, 2004] proposes another algorithm that improves the number of entries each chunk contributes to the index CAM.

2.2.2 Decomposition into Single Fields

Techniques for single field searches have been studied for a long time. It is a natural approach to decompose the multiple field search into a sequence of single field searches. For hardware implementation, this immediately creates a significant chance for parallelization as in most of the methods, their stages may be pipelined.

The main issue in combining single field search approaches is how to aggregate the results efficiently. Stages of the lookup process can return sets of matching results that have to be further restricted. The goal is to limit the number of intermediate results by a reasonably small number.

2.2.2.1 Recursive Flow Classification

Recursive Flow Classification [Gupta and McKeown, 1999] is based on "numbering classification classes." Packet headers are split into logical chunks and values for the chunks are numbered. For example, if the filter tests port numbers 25, 80, $\geq$ 1024, we generate classes for the possibilities (we add a class for "any other number"). This results into four classes that can be encoded with two bits. In the following phase, we combine chunk classifications into all possible combinations recursively taking equivalences into account.

Classification with this method takes chunks of packet headers and finds class numbers for their values. The numbers serve as memory indexes to find combined classes. In the simplest case, mere concatenation of the class numbers can work satisfactorily; other ways to combine the numbers are possible. The classification process results into a single number denoting the identification of the class.

The main drawback of the method is large preprocessing time and memory requirements for large classifiers.

2.2.2.2 Bit Vector Approaches

Bit vector methods are targeted to a hardware implementation. Lakshman and Stiliadis [Lakshman and Stiliadis, 1998] developed a method called Parallel Bit Vectors. Assume that filters are sorted according to their priorities. The filters are viewed geometrically and each filter (a hyperrectangle) projects to an axes of the k-dimensional space. Beginning and end points of the hyperrectangles define elementary intervals.

Each elementary interval has an assigned m-ary bit vector where m is the number of filters. Each bit corresponds to a filter, sorting filters by priority. The vectors have ones in positions where the filter overlaps the associated elementary interval. Those structures are constructed for all dimensions independently.

Searching is done separately for each dimension, obtaining bit vectors. After AND-ing bit-wisely all the vectors, the first '1' bit denotes the highest-priority matching filter.

The scheme can be improved taking into account statistical observations on real filter set: the bit vectors tend to be sparse. Baboescu and Varghese [Baboescu and Varghese, 2001] introduced the Aggregated Bit Vector algorithm that partitions the bit vectors into chunks and only chunks containing ones are stored.

2.2.2.3 Crossproducting

Crossproducting [Srinivasan et al., 1998] is based on the following. Filter database is sliced into columns. Each column contains all distinct prefixes of the particular field. A packed is classified separately in each column. To combine the results, we build a table of crossproducts, i.e., we precompute the result for each possible combination of column results. The exponential memory needed for the basic scheme can be reduced by precomputing the results lazily and keeping a limited amount of them in a cache.

2.2.2.4 Distributed Crossproducting of Field Labels

Distributed Crossproducting of Field Labels (DCFL) [Taylor and Turner, 2005] is based on two observations. First, a number of unique filter fields that match a given packet is limited in real rule sets. Second, number of combinations of unique filter field values that match a packet is small.

The method is intended for parallel hardware, therefore search engines for filter fields are distributed. Moreover, each field can be searched using a special method suitable for that task. For each filter field, we list all distinct values and assign numbers to them. To combine result for the first pair of fields, we construct a crossproduct table containing only pairs that occur in the rule set. Then we create similar combination for the first pair and the third field, etc.

Classification of a packet starts with classifying its fields separately (and possibly in parallel). Each field returns a set of matching filters. We take the first pair and we remove all impossible pairs (i.e., we test them for set membership against the set of possible values). We continue the same way until all sets are combined.

2.2.3 Tuple Search

We will describe the basic method by Srinivasan, Suri, and Varghese [Srinivasan et al., 1999] called Tuple Space Search.

A tuple is defined by the number of specified bits in each field. Tuple search is based on the observation that number of distinct tuples is significantly smaller than number of filters in the rule set.

For addresses, the number of specified bits is defined as the number of non-wildcard bits in the prefix. For protocols, we set it to '1' if and only if the protocol is specified. Port ranges are less straightforward. Let us suppose that all port specifications in the filter set are not overlapping. We can then build a "nesting structure" of port ranges. Level of nesting is used as the tuple value, number of the range in a given level identifies the range.

All filters that map to a particular tuple share a mask, i.e., they have the same IP prefix lengths, etc. We can concatenate the required number of bits from each field to form a hash table. Testing all filters mapped to the tuple requires concatenating prescribed number of bits and checking the hash. Theoretically, all the tuples can be checked in parallel (although it is not easy in practice as the number of tuples is not known in advance).

Other improvements are possible, reducing the number of tuples that have to be searched exhaustively (e.g., Pruned Tuple Space Search). This approach is probably not easily scalable for IPv6 because of the expected length of address prefixes.

2.2.4 Decision Trees

A simple approach to classification on multiple fields is construction of a decision tree. Leaves contain filters or their subsets. To classify a packet, we use its headers as a search key. Leaves contain best matching filter. Construction of decision trees is complicated by the fact that filters contain several types of searches. Typically, all the types are converted into one that is used in the structure.

Some of the algorithms based on decision trees are called "cutting" algorithms. These algorithms understand packet headers as points of a *k*-dimensional space.

Grid-of-tries we discussed in Section 2.1.1.6 can be understood as a decision tree method. This scheme was believed not to be easily extensible to more fields than two [Gupta and McKeown, 1999]. Baboescu et al. developed a method called Extended Grid-of-tries (EGT) [Baboescu et al., 2003] that supports classical 5-tuple classification.

In following sections, we will describe other examples of cutting algorithms.

2.2.4.1 HiCuts

Hierarchical Intelligent Cuttings (HiCuts) by Gupta and McKeown [Gupta and McKeown, 2000] view the classification problem geometrically. HiCuts preprocess the filter set to build a decision tree that contain a small number of filters (up to a threshold). All tests are converted into range matches in order to avoid replication. Each node is cut into equal-sized partitions along a single dimension.

Headers of a packet are used to traverse the tree. When a leaf is reached it is searched linearly. Several heuristics for minimizing the tree depth are described in the paper, such as minimizing the reuse of child nodes, eliminating redundancies in the tree, etc.

Very similar approach was developed independently by Woo [Woo, 2000].

2.2.4.2 FIS Trees

Fat Inverted Segment (FIS) Trees [Feldmann and Muthukrishnan, 2000] is a framework for packet classification using independent field searches. Projections of the *k*-dimensional hyperrectangles to the "edges" define elementary intervals. An FIS Tree is a balanced *t*-ary tree. Each node stores a set of ranges. Leaves correspond to elementary intervals on the edges. The union of the range sets of the nodes visited on the path from the leaf node associated with the elementary interval covering a value to the root is the set of ranges that contain the point.

Building this structure starts by building the FIS Tree on one axis. For each node that contains a non-empty set of filters, we build an FIS Tree for the following axis in the search. During the search, we start by finding the elementary interval covering the first packet field and continue following pointers to the sets of elementary intervals covering projections of the following fields. We remember the highest-priority result found, it is the result when the search terminates.

2.2.4.3 HyperCuts

HyperCuts classification scheme [Singh et al., 2003] is, like HiCuts, based on a decision tree structure. Each node in the HyperCuts represents a k-dimensional hypercube. A node in the decision tree represents a decision taken on the most representative dimensions. It allows to simulate several cuts of HiCuts in a single step.

The pre-processing algorithm identifies fields (i.e., dimensions) with highest number of distinct elements. For each such dimension it determines the number of cuts to be performed. The decision is based on heuristics. It is based on a trade-off between the memory size available and the depth of the tree. (Balancing the two quantities is possible.) Finally, the structure is cut. The number of node's children depends on the number of dimensions in the cut.

The algorithm may be refined by node merging, removing overlapping rules that produce unreachable parts of the structure, pushing common rule subsets upwards (if all child nodes contain an identical subset of rules the subset can be moved to the parent node), and similar methods.

2.2.5 Decision Diagram Representations

The motivation to develop classification methods we have described so far is primarily to make lookups more efficient, to increase throughput and minimize latency. Another issue appears in practice when working with lists of filtering rules: the lists become more complex and therefore more difficult to understand. Filters are often maintained by several people. The effect of changing, deleting, and/or adding a rule may not be obvious. This issue is addressed by representing filter lists in a way that allows formal checking.

In general, decision diagram is a special graph that leads the search through a series of tests resulting in a terminal node that contains the result.

2.2.5.1 Binary Decision Diagram

Let us define the basic Binary Decision Diagram (BDD) structure and its special types [Bryant, 1986].

Binary Decision Diagram (BDD) BDD is a rooted directed acyclic graph with

- one or two nodes of out-degree zero labelled 0 or 1, and
- a set of variable nodes u of out-degree two. Variable $var(u)$ is associated with each node and the outgoing edges are given by functions $high(u)$ and $low(u)$.

Ordered BDD (OBDD) A BDD is *ordered* (OBDD) [Bryant, 1986] if on all paths through the graph the variables respect a given linear order $x_1 < \ldots < x_n$ and[4]

- (uniqueness) no two distinct nodes u and v have the same variable and low- and high-successor, i.e., $var(u) = var(v)$ and $low(u) = low(v)$ and $high(u) = high(v)$ implies $u = v$, and
- (non-redundancy) no variable node u has identical low- and high-successor, i.e., $low(u) \neq high(u)$.

It can be shown that for any Boolean function exactly one OBDD exists that represents it. Moreover, Bryant gives a procedure to transform a BDD into the ordered form. Unfortunately, it can result into an exponential increase in number of nodes.

Hazelhurst et al. [Hazelhurst et al., 1998], [Hazelhurst et al., 2000] propose a method to convert a rule set into an OBDD.

Several technical steps must be taken. Numbers are represented as bit vectors. For example, representing the 4-bit number we use the bit vector $\langle x_3, \ldots, x_0 \rangle$, each x_i is a Boolean variable. Testing $x = 3$ means $\langle x_3, \ldots, x_0 \rangle = \langle 0, 0, 1, 1 \rangle$ that results in Boolean expression $\neg x_3 \wedge \neg x_2 \wedge x_1 \wedge x_0$.

We assign each protocol a number $0, \ldots, n_p - 1$. The numbers can be represented by $m_p = \log_2 n_p$ bits, so we use variables $\pi_0, \ldots, \pi_{m_p - 1}$ to encode them. Similarly, we use variables for bits of addresses and ports. Checking address prefixes is obvious, we just test up to the mask length. Port range queries are transformed using equivalence

$$p \leq n \Leftrightarrow \bigvee_{i=0}^{n} p = i$$

into logical expressions.

The whole rule set can be defined recursively (describing also immediately the formulae):

[4] BDD satisfying uniqueness and non-redundancy is sometimes called *reduced* BDD. Moreover, when people speak about BDDs they actually mean OBDDs. One has to be very careful about definitions.

- If the rule set is empty, no packets are accepted so the Boolean expression is False.[5]
- If the first rule is accepting then the packet will be accepted if it matches the rule or if it is accepted by the rest of the ruleset.
- If the first rule is rejecting then the packet will be accepted if it does not match the rule and it is accepted by the rest of the ruleset.

The size of the OBDD is dependent on variable ordering. Problem of finding an optimal ordering is known to be NP-complete [Bollig and Wegener, 1996], in practice, heuristic routines for dynamic variable ordering are employed.

Although it is beyond scope of this thesis, we briefly mention interesting ideas on testing and verification of packet filters using the BDD representation.[6] One possibility to validate is to ask 'what if' questions (e.g., Do we accept packets on port `25`? What packets do we accept from network `147.251.54.0/24`?). The queries can be expressed as Boolean conditions and tested against the set. Even changes in the rule set can be evaluated. If the original set and the changed set are represented as BDDs R_1 and R_1, queries can be coded as conditions. For example, "are there packets accepted by R_2 but not R_1" as $\neg R_1 \wedge R_2$.

Sinnappan and Hazelhurst [Sinnappan and Hazelhurst, 2001] describe an approach to convert BDD representation of a packet filter onto an FPGA. The FPGA is intended to work as a coprocessor of a network adapter.

The ruleset is directly implemented as circuit classifying packets, using reconfigurability of FPGAs. The Boolean expression logic is first converted to a circuit logic definition (VHDL in this case). The circuit is mapped onto the FPGA using commercial software (and one should keep in mind that this step may be time consuming, in order of tens of minutes). The resulting FPGA reads the packet information and generates either 'accept' or 'deny' output signal. Full details including a formal model can be found in Sinnappan's thesis [Sinnappan, 2001].

2.2.5.2 Interval Decision Diagrams

Although the BDD-based schemes work well in specialised hardware, granularity of their decisions is too subtle to suit general purpose processors. Processors have significant overhead to process a single bit a time, reading a word containing several bits—depending on architecture—is necessary. Moreover, extracting a single bit causes extra overhead both in processing and storage.

[5] Assuming deny policy as the default.

[6] An interested reader may see, e.g., papers [Mayer et al., 2000], [Eronen and Zitting, 2001], or [Al-Shaer and Hamed, 2004].

In order to avoid this drawback, Christiansen and Fleury [Christiansen and Fleury, 2004], [Christiansen and Fleury, 2004] focus on using an Interval Decision Diagram (IDD) [Strehl and Thiele, 1998b] to perform classification. IDDs classify on integer numbers instead of mere bits, each node is associated to an interval. Each edge is linked to another node or to a Boolean terminal.

Interval Decision Diagram (IDD) We define an IDD node first. Let x be an integer variable defined on domain $D_x \subseteq \mathbb{N}$ and t a logic formula on integer variables. Let $\{I_i\}$ for $0 \leq i \leq k$ be a partition of D_x.
We call t an *IDD node* if one of the following holds:

1. $t \in \{True, False\}$

2. $t = (x \in I_0 \wedge t_0) \vee \ldots \vee (x \in I_k \wedge t_k)$,

where t_i are IDD nodes. We denote $t = x \rightarrow (I_0, t_0) \ldots (I_k, t_k)$. From the "processing point of view" this means if the variable x belongs to interval I_i then traverse to node t_i.
Let $var(t)$ be the variable tested on node t, i.e.,

$$var(t) = \begin{cases} x & \text{if } t = (x \in I_0 \wedge t_0) \vee \ldots \vee (x \in I_k \wedge t_k) \\ t & \text{if } t \in \{True, False\}. \end{cases}$$

We call an IDD node a *root* if it has no predecessor. Finally, IDD is a set of IDD nodes with just one root. An IDD is called *ordered* if on all paths through the graph the variables respect a given linear order.

All usual logical operations (negation, and, or, etc.) can be performed on IDDs. It allows to encode filters as IDDs and manipulate them through Boolean algebra. Optimisation can be performed on resulting IDD, like interval merging and/or pruning nodes with the only outgoing edge.

The remaining problem is how to transform packet filters into IDDs. The approach of Christiansen and Fleury is as follows. Let H be the finite set of all possible headers and $\Pi = \{accept, deny\}$ the set of policies. A *rule* is a pair of a set of headers and a policy: $r = (h, \pi)$ where $h \subseteq H$ and $\pi \in \Pi$.

We define a *filter* φ as a set of rules over $2^H \times \Pi$

$$\varphi = ((h_1, \pi_1), \ldots, (h_n, \pi_n)).$$

By extension, the filter can be understood as a function that maps headers to sets of policies, i.e., $\varphi \colon H \rightarrow 2^\Pi$. The function will be defined as

$$\varphi(p) = \{\pi_i \in \Pi \mid p \in h_i\}.$$

We define an *unambiguous filter* as a filter where the system of packet headers $\{h_1, \dots, h_n\}$ is a partition of H. An unambiguous filter can be transformed into an equivalent IDD. Unfortunately, Christiansen and Fleury do not provide a description of the algorithm; they only show several examples.

As real-world packet filters are not unambiguous, a transformation to ensure this property is needed. Let us first formalise first match filters. We say that a filter $\psi = ((h_1, \pi_1), \dots, (h_n, \pi_n))$ with a relation $<$ such that

$$(h_i, \pi_i) < (h_j, \pi_j) \iff i < j$$

is an *ordered filter*.

We extend the definition of the ordered filter into a function $\psi\colon H \to \Pi$:

$$\psi(p) = \{\pi_i \in \Pi \mid p \in h_i \text{ and } p \notin h_j \text{ for } \forall j < i\}.$$

Now we show that for any ordered filter ψ we can build an equivalent unambiguous filter $\psi' = ((h'_1, \pi'_1), \dots, (h'_n, \pi'_n))$, i.e., $\psi(p) = \psi'(p)$ for any $p \in \Pi$.

The filter ψ' is given by

$$\begin{aligned} \pi'_i &= \pi_i \\ h'_i &= h_i - \bigcup_{j<i} h_j. \end{aligned}$$

It can be easily seen that this filter is equivalent to ψ and unambiguous.

When classifying real world packets, more than just two resulting actions are needed. The IDD scheme can be extended with a richer set of actions and the formalism of Multi-Terminal IDDs can be used for that. Formally, we may replace the set of Boolean terminals with a set of actions and enrich all operations consequently. For formal details, we again refer to the paper [Christiansen and Fleury, 2004] where MTIDDs are computed out of IDDs for individual actions from the set.

The ideas have been implemented in Compact Filter [Compact Filter, 2005] for Linux. The Compact Filter is based on the same kernel interface as Netfilter so the user may compile it as a module. The IDD is precomputed by a user space control program and the kernel is fed with the filtering code.

3 Routing, ARP, and Packet Filtering in Software Routers

We could, of course, use any notation we want; do not laugh at notations; invent them, they are powerful. In fact, mathematics is, to a large extent, invention of better notations.
– R. P. Feynman: Lectures on Physics I, 17-5

This chapter introduces a formalism we designed to denote routing, layer 3-to-layer 2 address translation (ARP), and packet filtering. We will use the formalism developed here to describe combining routing and ARP and proving the method correct in the following chapter. In Chapter 5, we will add packet filter to the structure and show how the result can be employed in the hardware lookup engine.

We would like to excuse the fact that some parts of this chapter describe well known principles of packet processing. It is intended to recall the principles shortly and to explain the notation that puts the principles onto formal basis.

In order to process a packet, a router must be able to handle three essential questions:

1. whether to send the packet,

2. where to send the packet,

3. how to send the packet to the next hop.

The question "where" is answered by a routing table and it is a network layer decision. We will discuss this topic in Section 3.1. "How" is handled by the link layer—a link layer address of the next hop must be found, a translation mechanism between layer 3 and layer 2 addresses is needed (Section 3.2). Last but not least, "whether" is the work of a packet filter (Section 3.3). The world of packet filters is very rich. As opposed to routing and ARP where the ideas behind them have direct reflection in implementations, stating a useful formalism for packet filters is a delicate question. We will show that classical textbook definitions are not sufficient to build formal descriptions, and we argue that studying real-world packet filters is necessary to state the formalism reasonably.

3.1 Routing

To decide where to send a packet, the router consults its *routing table*. We build formal notation to describe routing tables in this section. The control process—routing protocol—is beyond scope of this work; we concentrate on the process of finding the *next hop* for the packet.

As the principles of routing are equal for both IPv4 and IPv6 protocols, we do not distinguish between them unless necessary. Results of this chapter are applicable for both of them. We must keep in mind that routing tables for the protocols are distinct and they are processed separately. We also use terminology of IPv4 as it is widely known.

3.1.1 Routing Table

Definition 3.1 *IP*, **routing table**
Let *IP* be the set of *IP addresses*. *Routing table* is a function of IP addresses returning a record denoting where to send the packet

$$R\colon IP \rightarrow Interfaces \times IP.$$

The *Interfaces* is a finite set of *network interfaces* of a particular router. The result of routing is a pair consisting of the output interface and the IP address of the next hop.

Network interfaces (usually connected to a single physical link or several of them) are connected to a network. The network shares the most significant bits of the address, the common part is called a *network prefix* and its length is a *network mask*. Usual notation is, e.g., `147.251.54.0/24`.

Basic principles of sending packets over a subnetted network have been stated in [Mogul and Postel, 1985].[7] To handle a packet, a router checks its destination address and finds the *longest matching prefix* in its routing table.

An example of routing table is in Figure 3.1. Packets destined to the network `147.251.54.0/24` are sent through `eth0` interface. The address of the next hop will be directly the final destination of the packet. Any other packet goes to the router that connects this network to the outside Internet, called a *gateway*. Its next hop IP address will be the address of the gateway.

[7] An interested reader can also learn about the motivations in [Mogul, 1984].

```
Destination   Gateway       Genmask         Flags  Iface
147.251.54.0  0.0.0.0       255.255.255.0   U      eth0
0.0.0.0       147.251.54.1  0.0.0.0         UG     eth0
```

Figure 3.1 Example of a routing table

Rows of the routing table are essentially of two kinds, direct and indirect. We will characterise them with several equivalent conditions.

Definition 3.2 Direct and indirect routes
Direct route is a route leading to the network the router is directly connected to (a *local network*). A packet sent to this network is destined to a host connected to the same subnet as the output interface of the router, sharing the same network prefix with the interface. Its next hop address is equal to its final destination.

Indirect route is a route to the network that is accessed via an intermediate router, a gateway. The gateway is the next hop of the packet. Indirect route is a route that is not direct.

3.1.2 Formal Notation

Although we have described routing tables as longest matching prefix structures, we define it as first match lists. We will show later in this section that both representations are equivalent. To describe routing tables, we use the following notation.[8]

Definition 3.3 Routing table syntax
A *routing table* R is a sorted list of Size(R) rules. *Route* R_i is the i-th rule for $1 \leq i \leq$ Size(R) and Len(R_i) is the length of the prefix in that rule.

Let $[R_i]$ (which is a subset of IP) stand for the packets that match the i-th rule. If the routing table is represented with a trie structure, $[R_i]$ corresponds to the subtrie addressed by the prefix of R_i. Len(R_i) is then the length of the prefix[9]. The prefix is said to have *full length* if it contains a complete IP address (i.e., Len(R_i) = 32 for IPv4 and Len(R_i) = 128 for IPv6).

Route R_i is called *default* if Len(R_i) = 0 (note that $[R_i] = IP$ for the default route). On the "output side," $\mathrm{NH}_{\mathrm{IP}}(R_i)$ is the IP address of the next hop and $\mathrm{NH}_{\mathrm{Int}}(R_i)$ is the output interface of the i-th rule.

[8] The syntax is strongly inspired by the work by Frantzen [Frantzen, 2003].
[9] In a non-compressed trie it corresponds directly to the length of the path from the root to the node.

The routing table does not have to cover the complete address space. If a destination address is not found in the routing table, the router drops the packet and informs the sender of the packet with "no route to host" error message. This situation never occurs when the default route is defined.

3.1.3 Properties of Routing Tables

To make further processing easier, we require that the routing table satisfies several conditions. We have two types of conditions. First of them serves to restrict the model to real routing tables, the other to make further processing easier and/or possible. The requirements go without loss of generality.

1. (First match) The rules are sorted in non-increasing prefix lengths, therefore more special rules precede more general ones. Formally, for $1 \leq i < j \leq \mathrm{Size}(R)$, condition $\mathrm{Len}(R_i) \geq \mathrm{Len}(R_j)$ holds.

2. (Uniqueness) Moreover, prefixes of the same lengths are disjoint, i.e., for all i and j such that $1 \leq i < j \leq \mathrm{Size}(R)$ and $\mathrm{Len}(R_i) = \mathrm{Len}(R_j)$ we have $[R_i] \cap [R_j] = \emptyset$. Hence at most one longest matching prefix exists for each destination address.

The ordering allows us to define the semantics meaningfully.

Definition 3.4 Result of routing
To obtain the result of routing for a destination address $p \in IP$, we find $R(p) = R_i$ where i is the smallest index satisfying $p \in [R_i]$. Due to the ordering, it corresponds to finding the longest matching prefix of the destination address p.

Note that the next hop address $\mathrm{NH_{IP}}(R(p))$ is the address of the gateway for indirect routes. For direct routes, $\mathrm{NH_{IP}}(R(p)) = p$ as the next hop is the final destination of the packet on local network.

In practice, a routing table is usually kept in a variant of trie data structure. We gave a survey of such structures in Section 2.1.1. The longest matching prefix has to be found in trie structure. We will use two representations in this thesis: a longest-match trie and a first-match list that corresponds directly to our definitions. It is necessary to show that both representations are equivalent.

A first-match list equivalent to a given trie structure can be obtained by means of traversing the trie from longest prefixes. The routing table contains at most one outcome for a particular prefix, therefore the first match and uniqueness properties are satisfied. For the opposite conversion, prefixes from the list are just inserted into a trie. This is possible as prefixes of equal lengths are disjoint and the best matching prefix in the trie for an address is the first matching rule from the list.

We will need an extra property to be satisfied by the routing table. It is not related to the routing table representation and it is not intended to model real world routing tables.

3. (Relationship of direct and indirect networks) Direct networks do not contain indirect networks as their subnets with the exception of full length entries. Formally, for each j such that $1 \le j \le \text{Size}(R)$ and R_j is direct we require that if R_i exists for an i such that $1 \le i < j$ and $[R_i] \cap [R_j] \neq \emptyset$ then R_i is also direct or R_i is a full-length entry. (Note that $[R_i] \subset [R_j]$ in both cases.)

This property is purely technical. It will be necessary to allow us to re-arrange the order of records when routing and ARP tables are combined together in Section 4.2. We will be able to show its application there, mainly to prove Lemma 4.6.

Using this property would be easier if we omitted the possibility that the R_i record may have full length. On the other side, full length records may be quite common therefore the goal was to find as weak condition as possible.

As this requirement does not have to be satisfied by real-world routing tables, we will demonstrate how routing tables can be converted to comply with it without change of semantics. We will show the method of conversion in the following Section 3.1.4.

3.1.4 Prefix Expansion

The easiest way may be to expand the problematic indirect prefix to a set of full length records. This leads to an increase of routing records that is exponential to the address length, more precisely to the address length minus the length of the expanded prefix. It makes this method feasible only for very long indirect prefixes and absolutely unacceptable for IPv6 where the lowest 64 bits are expected to be an interface address.

A practically usable method is to expand the prefix of the direct network so that the conflicting indirect entry is no longer in its subnet. Because of the prefix expansion, it is easier to formulate and prove the method using trie representation of the table. We write that string a is a prefix of b as $a \lhd b$. Then, the property 3 from Section 3.1.3 can be re-written as follows. We require that if the routing table contains a direct prefix a and a prefix b such that $a \lhd b$ then b is either direct or b has full length.

We show now how to solve a single occurrence of violation of this property by means of prefix expansion. Then we show how this relationship of prefixes can be efficiently detected.

Let us have binary prefixes $a = x_1 \dots x_k$ and $b = x_1 \dots x_k y_1 \dots y_l$ (where $x_i \in \{0, 1\}$ and $y_i \in \{0, 1\}$). The prefixes represent address spaces. Length of prefix a is k, length of b is $k + l$. The task is to expand the prefix a into a set of prefixes that

Input: routing table R, prefixes $a = x_1 \ldots x_k$ and $b = x_1 \ldots x_k y_1 \ldots y_l$ (present in R) such that $l \geq 1$.
We construct prefixes

$$\begin{aligned} a_1 &= x_1 \ldots x_k \neg y_1 \\ a_2 &= x_1 \ldots x_k \;\; y_1 \neg y_2 \\ &\vdots \\ a_l &= x_1 \ldots x_k \;\; y_1 \;\; y_2 \ldots \;\; y_{l-1} \neg y_l \end{aligned}$$

We construct a set $I = \{i \mid i \in \{1, \ldots, l\}$ such that a_i is not in $R\}$. Routing table R' is the copy of routing table R with prefix a removed and prefixes a_i added with the same outcome as a had for all $i \in I$.

Algorithm 3.1 Expanding a prefix in a routing table

covers the same address space and they are not prefixes of b. The expansion is based on the fact that a can be expressed as $a_0 = x_1 \ldots x_k 0$ and $a_1 = x_1 \ldots x_k 1$ (this relationship is usually called Shannon expansion). We take $a_i \in \{a_0, a_1\}$ such that a_i is not a prefix of b, put a_i into the table instead of a in case that it has not been already present and continue expanding the other one. We never overwrite prefixes in the table during the process. The formal description is given in Algorithm 3.1 and illustrated in Figure 3.2.

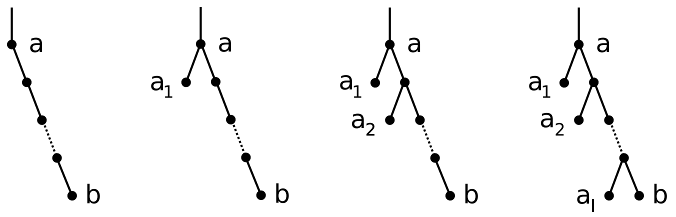


Figure 3.2 Algorithm 3.1 illustration

We have to show that the result of the Algorithm 3.1 discards the unwanted relationship of the prefixes. The following lemma is easily seen from the form of the prefixes generated.

Lemma 3.5
Prefixes a_i constructed in Algorithm 3.1 are not prefixes of b.

Proof
Prefixes a_i differ in the final negated bit from b. □

We have to show that the new routing table behaves equivalently with the old one.

Theorem 3.6 Correctness of the expansion
The result of longest matching prefix lookup for an address $p \in IP$ is the same for routing table R and for table R' generated by Algorithm 3.1.

Proof
Let $p \in IP$. It either is or is not routed by the prefix a in table R. Let us start with the latter case.

If p is not routed by the prefix a in R, it either has not prefix $x_1 \dots x_k$ at all and nothing has changed for it in R' or it has been routed by a prefix c such that $a \lhd c$, but c is left untouched in R' as the algorithm never changes such prefixes.

The remaining case is that p is routed by the prefix $a = x_1 \dots x_k$ in R. Then p itself has prefix a and no other record with prefix a exists in R that would be a prefix of p (otherwise it would be routed by the longer prefix).

We proceed by induction to l (i.e., the difference of lengths of a and b):

- Let $l = 1$. If p had a prefix $x_1 \dots x_k y_1$ then it would be routed by b. Hence it must have prefix $x_1 \dots x_k \neg y_1$.

- Let the theorem hold for $l - 1$. Then p is routed either by some of $a_1, \dots, a_{l-1}$ or by $a_l = x_1 \dots x_k y_1 \dots y_{l-1} \neg y_l$. The remaining possibility is that it would be routed by $x_1 \dots x_k y_1 \dots y_l = b$ which is not possible as we supposed p not to be routed by b.

Therefore the result of routing is the same for both R and R' as all the prefixes a_i have the same outcome as prefix a. □

Violations of the property 3 on page 36 are easy to detect traversing the trie containing the routing table in Depth-First Search[10] (DFS). Returning back from an indirect non-full-length node during the DFS, we just check that each node on the path back to the root is not direct. If it is not the case, we have to expand the prefix using the method above.

The DFS algorithm never returns to a node it has already visited in the trie [Cormen et al., 2001]. We perform the expansion only *returning* from the node that shall be expanded. The expansion changes only the subtrie of the node. Taking those facts into account, we may continue the DFS after the node is expanded in order to detect and expand other problematic prefixes. Hence the only Depth-First Search through the whole trie is needed to find and eliminate all prefixes violating the property 3.

[10] For a detailed description of DFS properties see [Cormen et al., 2001].

Theoretically, the prefix expansion described in this section may add at most vw prefixes where v is the number of direct networks in the routing table and w is the address width. On the contrary, tunnelling is the only practical situation that may cause the unwanted relationship of prefixes. Supposing that number of tunnels configured is typically small and number of direct networks is also limited, the increase in number of routing records is negligible. Therefore, for the purpose of complexity estimates, we will ignore the prefix expansion step; Size(R) can be understood roughly as routing table size when complexity is discussed.

We have shown that the properties stated in Section 3.1.3 have no effect on generality of the model. *From now on we suppose that routing tables satisfy the properties 1–3 from Section 3.1.3 unless said otherwise.*

3.2 Link Layer Addressing

Nodes of the network are connected with *links*. In order to reach the destination host, a packet must be moved over each link on the path. Link layer uses its own addressing, therefore an address translation mechanism is necessary. We prepare a formal description of the translation mechanism in this section.

An address on the link layer is usually called *physical address*, *LAN address*, *hardware address*, in the Ethernet context also *MAC (Media Access Control) address*. Their common property is that the addresses have flat structure (as opposed to a hierarchical structure of IP addresses) and an address should be permanent to a particular device and unique[11].

As most link layer protocols (like 802.11 wireless Ethernet, ATM, Frame Relay, etc.) use similar way of addressing, we can use Ethernet as a well-known and illustrative model and as a typical example. We will also use Ethernet terminology, although the principles cover a wide class of level 3 to level 2 address translations.

3.2.1 Address Translation Mechanism

Translation principles of network and link layer addresses are described in [Plummer, 1982]. The router maintains an *ARP table* containing IP and MAC address pairs for machines on the local network. The records are kept for a specified amount of time (typically several minutes), so the table is often called an *ARP cache*.[12] An example of ARP table is shown in Figure 3.3.

[11] At least in theory. Physical addresses used to be written in ROMs. Today's adapters keep their addresses in a flash-type memory, allowing experienced users to change them.

```
Address         HWtype  HWaddress           Flags Mask  Iface
147.251.54.1    ether   00:E0:81:27:DF:7B   C           eth0
147.251.54.10   ether   00:20:ED:5E:6D:98   C           eth0
```

Figure 3.3 Example of an ARP table

3.2.2 Formal Notation

Definition 3.7 ARP table
Formally, ARP table is a function

$$A\colon IP \to MAC$$

where *MAC* is a finite set of physical addresses.

In order to keep the notation consistent, *ARP table A* is a list of records. We denote *ARP records* as A_i for $1 \leq i \leq \mathrm{Size}(A)$. An ARP record A_i translates an IP address from $[A_i]$ to the MAC address. The ordering of rules is not relevant here as the set $[A_i]$ contains just a single IP address, not a portion of the address space denoted by a prefix. We require the records to be unique, i.e., for $1 \leq i < j \leq \mathrm{Size}(A)$ the condition $[A_i] \cap [A_j] = \emptyset$ holds. The result of the ARP lookup is the MAC address $\mathrm{NH}_{\mathrm{MAC}}(A_i)$.

The *result of ARP lookup* for a destination address $p \in IP$ is the record $A(p) = A_i$ such that $p \in [A_i]$.

The ARP table does not have to cover the complete IP address space. If the corresponding MAC address is not found for the next hop, ARP protocol is used to learn it. If it does not succeed, the packet is dropped and an error message is returned. For constructing tables for hardware lookups, only "current snapshot" of the table is necessary, as explained in Section 4.2. The mechanism of learning ARP records is purely matter of the operating system of the router, we therefore do not include the ARP protocol itself in the model.

[12] The structure is called *neighbour cache* in IPv6 and this term is sometimes used in IPv4 context, too. Anyway, the author preferred the older term in order to call routing-ARP-filtering structures RAF and not RNF which looks quite ugly.

3.3 Packet Filtering

The aim of this section is to state a useful formal definition of packet filters. We will discuss origins and motivation for the formalism. If a reader is interested in the formal syntax only, it is found in Section 3.3.5.

As the terminology in the literature differs, we start with basic definitions. A *firewall* is a combination of hardware and software that isolates an internal network from the outside Internet [Kurose and Ross, 2001], allowing some packets to pass and blocking others. More briefly, firewall is often defined as a box filtering network traffic [Cheswick et al., 2003]. Although various classifications appear in the literature, we usually distinguish two basic types of firewalls, application gateways and packet filters.

An *application gateway* is an application-specific filter that bases its filtering decision on scanning content of each packet. Application gateways usually need more extensive and complex processing of the packet body therefore are not very suitable for processing in our type of hardware. Application gateways depend on thorough knowledge of the application, often even on keeping state of the application level connection. Applications change rapidly compared to underlying network layers, development of such hardware supported application gateway would be inefficient in terms of the price of the development itself related to the performance gain obtained. An example of an application gateway may be scanning incoming mail for viruses and/or web proxy (especially with some "added value," say, enriched with blocking pictures of specified type).

Usual definition in the literature [Cheswick et al., 2003] says that *packet filters* work by discarding packets based on their headers. No context is kept, so the decision is based solely on the content of the current packet.

While formal model of routing and ARP tables has been based on principles, no single directly usable description of packet filters exists in the literature. The definition above is very illustrative but not completely adequate. It says that packet filters discard packets. Although it explains the principle to a student, it is not really true—packet filters often support much wider repertoire of actions. The definition says that the fate of packets is based on contents of their headers. Again, this is not very exact, most packet filters allow to use, e.g., the input interface as part of the filtering decision and the input interface is not part of headers.

When we cannot start with the "textbook definition," we shall study the literature. Papers on packet filtering can be divided into two categories. They either discuss implementations or some general principles of packet classification. Papers interested in principles typically just define packet filters to suit their purposes but they do not discuss whether the definition matches real implementations. We will do this work in the following. We will study and compare a representative set of real-world firewalls in order to

- state a useful formal description of packet filters to prevent the model to be unnecessarily weak,
- to show that the model matches the reality,
- and show that the filters can be converted into the hardware classification engine.

Following observations are based on a representative list of filters used in various operating systems and routers.

The filters turn out to have significant similarities that help us to treat them as "classes" during the survey. Anyway, the classification is based on subjectively chosen similarities and it is somewhat fuzzy—another classification is surely possible.

We may distinguish three basic "styles of filtering:"

- open source BSD systems:
  - pf packet filter [PF, 2005a][13] (originally for OpenBSD [OpenBSD, 2005] but also ported for FreeBSD [FreeBSD, 2005] and NetBSD [NetBSD, 2005]),
  - Darren Reed's IP Filter [IPF, 2005a] (available on FreeBSD, NetBSD, Solaris, etc.[14]),
  - FreeBSD-sponsored IPFIREWALL (IPFW) [IPFIREWALL, 2005], [Lidl et al., 2002],
- Linux:
  - ipchains [Russel, 2005a],
  - iptables/netfilter [netfilter, 2005a],
  - Compact Filter [Compact Filter, 2005] (its theoretical foundations are discussed in Section 2.2.5.2),
- commercial routers:

[13] Although pf stands for "packet filter" the pf is capable of NAT, port forwarding, and stateful rules, so it is a firewall in our terminology.

[14] OpenBSD used to incorporate a derivative of the IP Filter, but it was removed due to licensing problems. Other projects also seem to tend to pf. See `http://www.openbsd.org/lyrics.html` for details.

– boxes by Juniper Networks, Inc. [Juniper Networks, Inc., 2005a],

– boxes by CISCO Systems [Cisco Systems, 2005].

Moving towards a formal definition of packet filter, we divide the discussion into three basic parts: "inputs" of filtering process, the "way" how filtering is performed and "outputs," i.e., the set of possible results of packet filtering. During the survey, we will try to identify what differences are a matter of principle and what is just a syntactical construct with the same expression abilities.

Contemporary firewalls support many features not directly satisfying our definition of packet filter or firewall, such as network address translation (NAT), various packet rewriting and editing, including even very obscure features like changing the Time-to-Live value, etc. Those features are not supported by the hardware accelerator. They can be done by sending the packet to the operating system to be processed.

Each filtering tool has its peculiarities, fortunately often belonging just to the "syntax sugar" category. Anyway, it makes the comparison study quite difficult. We will not take such features into account if they are not parts of the basic functionality of such tool. Even the documentation is not always absolutely clean about the firewall setup[15], forcing the administrator to use the method of trial and error.

3.3.1 Inputs of Filtering

The first difference to the classical definition is that headers of packets may not be enough to decide what to do with them in the filter. Other information is often used, e.g., input interface the packet arrived, output interface the packet should be sent out, or other "implicit" observable properties ("observables" in terms of quantum physics) of the packet.

Packet information We will use term *packet information* for a set of fields found in packet headers (like source/destination addresses/ports, level 2 addresses, protocol type, etc.) and other observables of the packet (input interface, output interface). In formulae, symbol *PktInfo* stands for the Cartesian product of all packet observables.

The packet information corresponds to a data structure used internally by the operating system to describe a packet. What fields are accessible for a filtering tool

[15] E.g., [Juniper Networks, Inc., 2005a], page 161: "`discard`—The packet is not accepted and is not processed further. Discarded packets cannot be logged or sampled.", page 162: "The firewall stops logging discard and reject actions at a high traffic rate."

depends mainly on implementation of the structures in a particular operating system. From the hardware accelerator's point of view, packet information is stored in the Unified Header structure, see Section 1.2.2 for its description.

3.3.2 Evaluation Order of Firewall Rules

We will study evaluation order of firewall rules. The two main groups are first-match and last-match filters. Anyway, special features that change evaluation order can be found in filters. We will show how they can be avoided without change of semantics.

3.3.2.1 First-match and Last-match Filters

Reed's IP Filter and pf packet filter are typical examples of "last-match" filters. For each packet processed by the packet filter, the filter rules are evaluated in sequential order, from first to last. The last matching rule is remembered and used as the result. Moreover, keyword `quick` denotes that the rule is considered the last matching rule, and evaluation of subsequent rules is skipped.

The remaining filters we consider are essentially first-match, the first matching rule is taken for a packet. As a syntactical shorthand, blocks of rule sets can be named and used as a subroutine (called a chain in ipchains and iptables). Similar functionality can be achieved using `skipto` commands in IPFW. We can obtain purely first-match representation by expanding the named blocks (they have to be finite so the expansion has finite depth, otherwise the evaluation of the filter would be infinite, probably causing a kernel crash).

A first-match filter can be easily converted to last-match one either by reverting the order of rules or using the `quick` keyword for all the rules. A purely last-match filter that does not contain `quick` rules can be transformed to first-match again reverting the order of rules.

3.3.2.2 Last-match Filters with `quick`

More sophisticated conversion is necessary when the last-match filter contains `quick` keyword in some of its rules. Having rules[16] $F = \{F_1, \ldots, F_n\}$, we denote by $\langle F_i \rangle$ the packet information *(syntactically) matched* by the rule F_i (not taking into account the remainder of the rule set, just the i^{th} rule itself). Let $\langle F_i \rangle_F$ be the packet information that *semantically matches the rule F_i with respect to the rule set F*, i.e., it is just the packet information actually matched by F_i when processing the packet against the filter.

[16] The formal syntax will be defined precisely in Section 3.3.5. We only sketch the essentials to explain the conversion here.

In the last-match filter containing `quick` rules, the rule F_i semantically matches just packets that

- match the rule F_i,
- do not match any following rule F_j for $i < j \leq n$, and
- do not match any preceding `quick` rule F_k for $1 \leq k < i$.

Formally,

$$\langle F_i \rangle_F = \langle F_i \rangle - \left(\bigcup_{i<j\leq n} \langle F_j \rangle \cup \bigcup_{\substack{1\leq k<i \\ F_k \text{ is quick}}} \langle F_k \rangle \right).$$

The resulting rules can be represented in first-match semantics. To see that, we will show a stronger result—the rules are pairwise disjoint.

Lemma 3.8
Sets $\langle F_i \rangle_F$ are pairwise disjoint for $1 \leq i \leq n$.

Proof
Let $p \in \langle F_i \rangle_F \cap \langle F_j \rangle_F$ for $i \neq j$. Suppose without loss of generality that $i < j$. Then, from the definition, $p \in \langle F_i \rangle$ and $p \notin \langle F_l \rangle$ for any $i < l$. Specially, $p \notin \langle F_j \rangle$, hence $p \notin \langle F_j \rangle_F$ which is a contradiction. □

We can see from the construction that the behaviour of the resulting filter is equivalent to the original one. For each rule, we have just removed parts of *PktInfo* space that would never match when evaluating the rule in the whole ruleset anyway.

The scheme can be optimised by removing rules that are not satisfiable and by removing unreachable rules. Produced rules can be also simplified, mainly removing redundant tests and/or testing subsets of already tested fields.

Moreover, if the original filter was total[17], i.e., $\bigcup_{1\leq i\leq n} \langle F_i \rangle = \mathit{PktInfo}$, the resulting rules will be a partitioning of *PktInfo*. We have already shown that sets $\langle F_i \rangle_F$

[17] Totality of filters is a reasonable requirement: we want the filter to decide what to do with any packet coming through.

constructed are pairwise disjoint, so the remaining step is to demonstrate that $\bigcup_{1\le i\le n}\langle F_i\rangle_F = \mathit{PktInfo}$.

Lemma 3.9
Let $\bigcup_{1\le i\le n}\langle F_i\rangle = \mathit{PktInfo}$. Then $\bigcup_{1\le i\le n}\langle F_i\rangle_F = \mathit{PktInfo}$.

Proof
Let us evaluate the expression $\bigcup_{1\le i\le n}\langle F_i\rangle_F$.

$$\bigcup_{1\le i\le n}\langle F_i\rangle_F = \bigcup_{1\le i\le n}\left[\langle F_i\rangle - \left(\bigcup_{i<j\le n}\langle F_j\rangle \cup \bigcup_{\substack{1\le k<i\\ F_k \text{ is quick}}}\langle F_k\rangle\right)\right] =$$

$$= \bigcup_{1\le i\le n}\langle F_i\rangle - \bigcap_{1\le i\le n}\left(\bigcup_{i<j\le n}\langle F_j\rangle \cup \bigcup_{\substack{1\le k<i\\ F_k \text{ is quick}}}\langle F_k\rangle\right) =$$

$$= \mathit{PktInfo} - \bigcap_{1\le i\le n}\left(\bigcup_{i<j\le n}\langle F_j\rangle \cup \bigcup_{\substack{1\le k<i\\ F_k \text{ is quick}}}\langle F_k\rangle\right) = \tag{1}$$

$$= \mathit{PktInfo} - \emptyset = \mathit{PktInfo} \tag{2}$$

Equation (1) was obtained supposing the original filter is total. Equation (2) is based on the observation that just the i^{th} rule is missing in i^{th} disjunct in equation (1), so disjunctions of pairs of the expressions are empty. □

3.3.2.3 "Double Match" Features

Packet filters in routers by Juniper Networks find the first matching rule to handle a packet. Moreover, if the resulting action is marked with the `next term` action modifier, evaluation of the filter does not stop but continues against subsequent rules of the filter. Again, this does not affect expression abilities of the filter; we can expand the `next term` rule into a sequence of conjunctions of the rule with all the following rules (followed by the rest of the filter). Similar method can be used for first-match filters that allow performing "double match," e.g., IPFW `count` rule action which updates counter of packets matching the rule and the search continues in the rule set.

The principle of expansion is indicated in Figure 3.4 for a filter containing four rules, extending to more rules or multiple `next term` occurrences is straightforward. The second rule is marked `next term` (not taking into account the resulting actions). The rule is expanded into a series of rules matched by packets that would match the `next term` rule and a subsequent rule. The right part of the

Original filter	Expanded filter
rule_1	rule_1
rule_2 `next term`	rule_2 and rule_3
	rule_2 and rule_4
	rule_2
rule_3	rule_3
rule_4	rule_4

Figure 3.4 Expanding `next term` statement in first-match filters

scheme shows the expanded filter. We omit resulting actions in the scheme, they depend on the actual filtering tool. In general, it is necessary to study all combinations of actions originating from the rules. Note the second rule repeated below the expansion in the expanded filter—it is necessary for packets that match just the second rule and none of the subsequent ones.

3.3.3 Position of Filtering in IP Stack

Packet filtering may occur before routing, after routing, or both. Dividing filtering into several stages that take place in various stages of packet processing serves mainly to make administration more comfortable (and more confusing). Some filtering tools provide distinct filtering chains for packets passing through, getting into, and sent out of the router. Although the principles of filtering are very similar for all tools, an administrator has to be extremely careful about the precise meaning of the configuration.

3.3.3.1 Overview of Filtering Schemes

Packet filtering in open-source BSD platforms usually takes place in the input and output stages. Inbound filters affect all packets that enter the system, either destined there or to be forwarded. All the traffic leaving the system, generated by the system and/or forwarded, has to pass through outbound filters. A specification of an interface is an optional part of the filtering rule. This approach is used in OpenBSD pf packet filter, Darren Reed's IP Filter, and other (mainly BSD) systems. Having an input and an output filter in the system is considered to be the "standard BSD approach." Even though filtering is applied in two places of the IP stack, the filters usually do not require specifying the direction. In that case, the filter applies to both incoming and outgoing packets.

An administrator may even use several filtering tools at the same time on a single BSD box if he/she is curious enough to guess what effects it is going to cause.

FreeBSD-sponsored filtering application IPFIREWALL (IPFW) supports a bit richer set of filtering points. It allows to filter input and output traffic, in both cases in level 2 and level 3 of kernel IP stack separately. The points of filtering are controlled by kernel (`sysctl`) variables. The main difference between level 2 and 3 filters is that the level 2 headers are not available for the level 3 filter. Each packet is checked against the complete ruleset in all points the filter is established. If a rule matching patterns that are not accessible in the place of invocation is tested, it does not match. The documentation recommends to configure the rulesets separately for various points of filtering used in the configuration. As the filter configuration is global, this may be achieved by means of skipping to the appropriate part of the ruleset.

The purpose of splitting level 2 and level 3 filtering is a kind of optimisation: if the user needs only level 3 filtering it is not necessary to parse the packet including level 2 headers, to perform filtering, and to parse the headers again in order to push it to the higher level of the IP stack. Anyway, any level 3 filter can be used also on level 2.

Linux uses basically two filtering tools, ipchains and iptables/netfilter. The ipchains package is now considered obsolete and will be probably supported at most till Linux kernel version 2.8. Principles of packet processing in both the tools are very similar, user-visible changes are mostly syntactical, therefore we may restrict ourselves to iptables processing. In Section 2.2.5.2 we also mentioned Compact Filter. This tool uses the same kernel interface, it only compiles the filter into a more efficient form.

Iptables start with three lists of rules called INPUT, OUTPUT, and FORWARD by default. The IP stack first decides if the packet is destined for the host machine. In that case, the INPUT chain is applied and if it accepts the packet it is delivered to a process running on the host machine. If the packet is to be sent to another network node, it goes only through the FORWARD chain. Finally, packets generated by local processes traverse through the OUTPUT chain. The only difference of expression abilities is that INPUT filters can test the input interface of a packet (a rule specifying an output interface is syntactically correct but it will never match in the INPUT chain). Symmetrically, the OUTPUT chain can only test the output interface. In the FORWARD chain, both input and output interfaces may match. (It also means that FORWARD filtering necessarily takes place after routing when the output interface is known.)

Routers by Juniper Networks, Inc., distinguish data packets (packets to be forwarded by the router) and local packets (destined to the box or sent by it). Data packets can be filtered on devices equipped with the Internet Processor II only, local packets may be controlled on all routing platforms.

For each interface, a firewall can be applied to incoming traffic, outgoing traffic, or both. A named filter, consisting essentially of an input and an output filter, is bound to an interface. All traffic incoming through an interface is checked against the incoming filter for the interface (including local packets), local packets are moreover checked against the local input filter (formally bound to the loopback incoming interface). Processing of outgoing packets is symmetrical.

Filtering in commercially available routers by CISCO Systems is very similar to Juniper platform, in essence, first match filters can be configured separately for each input and output interface.

3.3.3.2 Expression Power Comparison

Based on the survey in the previous section, we will study what differences may affect the expression power of the filters.

Let us suppose for the rest of this section that all the filters we deal with have first-match semantics—we have shown in Section 3.3.2 that this does not affect generality.

The first difference is whether filters are defined "one rule-set per interface" or for all interfaces in a single list (allowing to test the interface but not requiring it). Both types can be easily converted. Let us suppose first-match semantics of the language and filtering rules that contain just conjunctions of terms. Having a separate filter for each interface, we obtain a global filter by adding "and interface *i*" explicitly to each rule and joining the rule lists into one. In the opposite direction, we may merely put the global filter to all interfaces. Moreover, rules containing "interface *i*" can be omitted from all lists except the list for the interface *i*.

We stated that having an input and an output filter is a common practice in BSD configuration. Not all BSD filtering tools have the rulesets strictly separated. For example, direction specification is mandatory in ipf for each rule (so the filter behaves as separated rulesets applied in two places of the IP stack). On the opposite, if direction specification is omitted in a pf rule, the rule applies to both input and output traffic (a single ruleset applied in two places). Using the same method as for per-interface filters, rules with optional and mandatory direction specification may be converted.

We have shown that defining rule-sets per interface or globally as well as requiring mandatory direction specification is just a "political decision" and matter of taste.

Another difference we observed in the previous section is in the "style of processing," either having a BSD style input and output filters and/or Linux style INPUT/FORWARD/OUTPUT chains.

To convert the BSD input and output filters into INPUT/FORWARD/OUTPUT, we copy the input filter into INPUT and output filter into OUTPUT. It ensures that locally destined and generated packets go through input and output filters.

The FORWARD chain is a bit more complex to build, we have to arrange it to test both input and output filters. From theoretical point of view, we can compute a Cartesian product of the input and output filters according to the scheme

$$\begin{array}{lcl}
\mathrm{input}_1 \text{ and } \mathrm{output}_1 & \rightarrow & \mathrm{action}_{1.1} \\
\mathrm{input}_1 \text{ and } \mathrm{output}_2 & \rightarrow & \mathrm{action}_{1.2} \\
\dots & & \dots \\
\mathrm{input}_1 \text{ and } \mathrm{output}_n & \rightarrow & \mathrm{action}_{1.n} \\
\dots & & \dots \\
\mathrm{input}_m \text{ and } \mathrm{output}_1 & \rightarrow & \mathrm{action}_{m.1} \\
\dots & & \dots \\
\mathrm{input}_m \text{ and } \mathrm{output}_n & \rightarrow & \mathrm{action}_{m.n}
\end{array}$$

Figure 3.5 Ordering rules in FORWARD filters

in Figure 3.5. The rows contain conjunctions of input and output rules. Considering only actions Accept and Deny, the final action is Accept if and only if both of its parts have Accept. For richer set of actions, all combinations must be evaluated, e.g., combination Accept, Log from the input rule and Deny from the output one should result into Deny, Log in the combination. Moreover, input rules resulting in Deny action do not have to be expanded using output rules as all of them would result into Deny anyway.

If both level 2 and level 3 input (resp. output) filters are specified in IPFW, the same method can be used to combine them into a single level 2 input (output) filter.

A "practical" approach to building the FORWARD chain can produce a more concise representation. We can insert the input filter into the FORWARD chain and to rewrite all accepting actions in the chain to calling a new FORWARD′ chain that contains the output filter. It simulates performing the input filter and then the output one. The FORWARD′ chain is stored just once, as opposed to the full table above.

To convert the Linux INPUT/FORWARD/OUTPUT scheme into BSD input and output filters, a way to describe addresses of the host machine is needed. This can be directly supported by the tool, for example, IPFW has a keyword `me` that matches any IP address configured on an interface in the system. It is expanded when evaluating the packet against the rule. If a similar feature is not supported, it may not be sufficient to build the list of host's addresses by hand. Unicast addresses do not change very frequently, on the other hand, multicast group membership may be quite dynamic, changing sets of interface addresses rapidly. We will use the symbol `me` to describe the addresses of the host (abstracting from how it is obtained).

The simplest conversion requires that the output filter is able to test both the input interface the packet arrived and the output interface prescribed by routing. Having the possibility to test both the interfaces in the output filter, we can create the input filter containing the rules from INPUT enriched with "and destined to `me`" to ensure that the INPUT rules match just packets destined to the host in the input filter. The output filter will contain the OUTPUT filter rules enriched with "and sent by `me`" followed by the FORWARD rules with "and not sent by `me`." Locally generated packets will therefore be tested by the OUTPUT filter, other routed packets by the FORWARD filter.

If it is not possible to test both interfaces in the output filter, we need to have another mechanism to simulate that. Tagging is a feature that allows adding a tag to the packet and to test it in later stages of processing. We can use it to remember the incoming interface in the input filter (just adding it after the rules for self-destined packets; it is possible as we have to tag only packets that are not destined to the host). We then rewrite all incoming interface tests in the OUTPUT filter just to testing appropriate tags.

Other methods of propagating the input interface value to the output filter may be supported by the tool. If it is not possible to test the incoming interface in the output filter, the input/output scheme would not be capable of processing the FORWARD rules containing the incoming interface test.

To sum up, we have shown that all considered filtering schemes have equal expression power and they can be converted one into another. The only exception is the iptables FORWARD chain. For that conversion, the input/output scheme has to support a way to test both input and output interfaces in a single rule and/or to simulate such test by other means and it has to support a way to describe own addresses of the router.

However, the hardware accelerator is capable of testing both interfaces in its processing: the input interface value is available in the Unified Header and the output interface is computed during the LUP search. In the following section, we will discuss position of the hardware accelerator in the system.

3.3.3.3 Position of the Hardware Accelerator in the System

In Chapter 1, we described that the accelerator behaves as a network interface card from the operating system's point of view (only "switching some packets by itself"). Taking that into account, we shall discuss the position of the accelerator in the system and state what kind of filters shall the accelerator incorporate. We will divide the traffic according to its destination to packets sent to and by the operating system and packets forwarded directly by the accelerator.

The accelerator can handle packets sent by the host computer operating system. The packet is inserted directly to the output queue of the appropriate interface, so no other processing is expected by the accelerator, it behaves just as an ordinary network interface card in this case. It determines also processing of the output filter—output filtering must be performed completely by the operating system, at least because no possibility of packet classification and filtering exists in the output queue.

Packets received by the accelerator may be either forwarded to another interface of the accelerator and/or passed to the host computer. Let us start with the latter case.

Packets passed to the host computer can be either destined there or they shall be forwarded through an interface that does not belong to the accelerator (e.g., a network interface card). Both the possibilities lead to identical actions for the accelerator, mere sending the packet to the operating system. In that case, the accelerator does not have to perform any filtering for such packets as they will be filtered by the operating system anyway. On the other hand, filtering them in the accelerator decreases the load of the system bus, especially when the ratio of dropped packets to the total traffic is high. It can be helpful in case of attacks to the host machine when the attacking data stream is stopped in hardware, preventing increase of system load and wasting bus bandwidth.

For packets forwarded by the accelerator, all the filters that would be applied in the operating system must be performed by the classification engine. It directly corresponds to Linux style FORWARD filter described in the beginning of Section 3.3.3.

3.3.4 Resulting Actions

The definition of packet filter speaks only about "discarding packets." Real-world packet filters support richer sets of actions, like accepting (which lets the packet go through), rejecting (drops the packet and informs the sender with an ICMP message), discard (drop silently), logging (write more or less detailed information about the packet to a log file), etc.

Although it is not always explicitly described in documentation of various filtering tools, results of filtering rules can be divided into two categories, basic actions and action modifiers. *Basic actions* determine whether the packet is accepted or discarded. Just one action is a mandatory part of each rule (or it has an implicit default value in some filters). *Action modifiers* add extra processing that may or may not change the packet. A rule can contain zero or more action modifiers.

For example, Junos operating system [Juniper Networks, Inc., 2005a] has basic actions[18] `accept`—accept the packet, `discard`—drop the packet silently, and `reject`—the packed is dropped and a rejection message is returned.

Following action modifiers can be used in Junos: `count`—add packet to the total count, `log`—the packet's header information is stored, `policer`—apply rate limiting, `sample`—sample the packet traffic, and finally `syslog`—log an alert for the packet.

[18] We omit actions and modifiers that are related just to evaluating of the filter and/or Junos specific processing, like setting forwarding classes and priorities.

Let *BasicActions* be the set of basic filtering actions of a particular filtering tool. In the simplest case, $\mathit{BasicActions} = \{\mathit{Accept}, \mathit{Deny}\}$. Let *ActionModifiers* be the set of action modifiers. Then, we define filtering actions as

$$\mathit{Actions} = \mathit{BasicActions} \times 2^{\mathit{ActionModifiers}}.$$

Although such syntax is useful to explain the concept of actions and modifiers, it would be technically unnecessarily complex to use—we would have to define functions to disassemble components of the product. To keep things simple, for $a \in \mathit{Actions}$, we will freely use syntax $a = \{\mathit{Accept}, \mathit{Log}\}$ (and $\mathit{Accept} \in a$ and $\mathit{Log} \in a$) for both actions and modifiers as misunderstanding is not possible. Of course, we should keep the concept of actions and modifiers in minds.

Processing of filtering rules in the accelerator is based on the observation that we can either process the packet in the accelerator or we have to send the packet to the operating system to be processed. When the packet is passed to the operating system IP stack, it is "always processed completely and correctly." We have no possibility of, say, performing accepting and logging so that we send the packet in hardware and pass it to the operating system in the standard way to write a message to the system log as no way exists to instruct the IP stack to log the packet but not to send it away without modifying the kernel IP stack code.

We divide the filtering actions into two groups: actions performed by the accelerator and actions requiring sending the packet to the operating system. We define a partitioning on *Actions* containing classes

- $\mathit{Actions}_{HW}$ of actions that can be performed by the accelerator,
- $\mathit{Actions}_{OS}$ of actions that have to be performed by the operating system.

No "absolute border" exists between the classes, they depend on the state of hardware accelerator implementation. E.g., if a mechanism for logging packets would be implemented in the accelerator (i.e., an interface for notifying the logging daemon through the driver), action "deny and log" can move to the hardware supported class.

3.3.5 Packet Filter Definition

Based on the survey and discussion of previous sections, packet filters have one feature in common: they partition the space of possible packet observables and attach actions to the partitions. Formally, packet filter is a pair (Π, A) where

- $\Pi = \{PktInfo_1, \dots, PktInfo_n\}$ such that $PktInfo_i \subseteq PktInfo$ for $1 \leq i \leq n$,
- Π is a partitioning of *PktInfo*,
- $A = \{a_1, \dots, a_n\}$ such that $a_i \in Actions$ for all $1 \leq i \leq n$,
- action a_i is assigned (and executed for) the set $PktInfo_i$.

Note that the definition does not specify how membership in a $PktInfo_i$ set is determined, it can be done by any computable function, e.g., matching a pattern on some fields of the packet header, performing a computation, or even an exhaustive list of matching packets.

Using this partitioning-based description directly is not very practical, it requires describing each subset explicitly. Therefore, we represent packet filters as first match lists. Such representation is equivalent to the partition-based description. A filter expressed as a partitioning can be directly interpreted as a first match list. A first match list can be converted to an explicitly expressed partitioning [Frantzen, 2003]. The basic idea of the conversion is that a packet accepted by the list must match an accepting rule and none of preceding rejecting ones.[19]

Definition 3.10 Packet filter
Let us define packet filter as a total mapping

$$F\colon PktInfo \to Actions$$

where *PktInfo* is the packet information (see Section 3.3.1) and *Actions* is a set of possible filtering actions (see Section 3.3.4).

A packet filter F consists of Size(F) rules. A filtering rule F_i is the i-th rule in F for $1 \leq i \leq \text{Size}(F)$. Let $\langle F_i \rangle \subseteq PktInfo$ be the set of packet information that matches rule F_i. We use $[F_i] \subseteq IP$ to denote the set of destination IP addresses that the rule matches. This notation is intended to keep the syntax compatible with our notation of routing and ARP tables.

The result of filtering for a packet information $p \in PktInfo$ is the record F_i for the smallest index i such that $p \in \langle F_i \rangle$. The resulting action is $\text{Action}(F_i) \in Actions$.

A packet filter must decide what to do with each packet, therefore we require packet filter to be *total*, i.e., to give a result for all packet headers. In practice, this is usually achieved using a *default action*[20] (often expressed as "default chain policy" and/or hardwired default behaviour of the filter). Rule F_i is called *default*

[19] We used similar conversion in Section 3.3.2.2 for evaluating last-match filters.
[20] Note that requiring the default action is a stronger condition than totality of the filter.

```
<fw-list> := <fw-rule> ";" ...
<fw-rule> := [<sif>] [<saddr>] [<sport>]
                   [<dif>] [<daddr>] [<dport>]
                   [<proto>] <action>
<action>  := "accept" | "deny" | "accept log" | "deny log"
<sif>     := "sif" <interface-id>
<dif>     := "dif" <interface-id>
<saddr>   := "saddr" <ip-address> "/" <network-mask>
<daddr>   := "daddr" <ip-address> "/" <network-mask>
<sport>   := "sport" <port-number> "-" <port-number>
<dport>   := "dport" <port-number> "-" <port-number>
<proto>   := "proto" <protocol-specification>
```

Figure 3.6 Grammar of packet filters

```
daddr 147.251.54.0/255.255.255.0 dport 80-80 accept;
daddr 147.251.54.10/255.255.255.255 accept;
deny;
```

Figure 3.7 Packet filter—an example

if $\langle F_i \rangle = \mathit{PktInfo}$. We suppose that the last rule in the filter is the default rule, i.e., $\langle F_{\mathrm{Size}(F)} \rangle = \mathit{PktInfo}$.

Without loss of generality, we suppose that filtering rules contain only conjunctions of atomic tests unless said otherwise. Rules containing disjunctions can be rewritten into sequences of conjunction rules in the first match filter.

To denote a packet filter, we will use grammar shown in Figure 3.6. An example of packet filter is depicted in Figure 3.7.

We have to describe how packet filtering interacts with routing and link-layer addressing in our model. The cooperation of routing and ARPs has been discussed at the end of Section 3.2. We incorporate packet filtering just after ARP resolution. The packet is routed, MAC address of the next hop is resolved, and packet filtering is performed. If the result of the filtering process is an accepting action then the packet is sent out.

4 Routing and ARP Table Combined

Parental advisory: explicit mathematical expressions.
– Miloš Liška

We have described routing, ARP, and packet filtering, and how they cooperate in the operating system. In order to perform equivalent lookups in the hardware lookup machine, combining them to a single lookup operation is needed. The method to combine lookups consists of two main parts: combining routing and ARP into a single structure and then adding the packet filter. The latter part of the method—adding packet filter—will be described in Chapter 5.

This chapter deals with combining routing and ARPs. The method is based on injecting ARP records to the routing table, creating a structure we call a routing-ARP table. Formally, we handle all the tables as first-match lists. We prove the method correct in the first match representation and we show how the first match routing-ARP table can be converted into longest matching prefix representation that discards possible redundancies.

Through the whole work, we suppose that the resulting structure will be interpreted in an environment where we can give up processing a packet and leave it to a "higher intelligence that always performs the correct action." Practically, the accelerator can send a packet to the operating system to be processed if handling the packet is not possible in the accelerator for any reason. We discuss this topic in the following Section 4.1. In the rest of this chapter (Section 4.2), we define the routing-ARP table (Section 4.2.2), describe and prove correctness of the method to combine routing and ARP into a single operation, resulting in the final algorithm presented in Section 4.2.5. In Section 4.2.6, we give a complexity estimate of the final structure.

4.1 Software Cooperation

An essential principle is used in the design: *If the hardware lookup cannot resolve a packet itself it can send it to the software in the same way as an ordinary network adapter would.*

In the algorithms, this will be denoted by a '*SW*' action that means the packet is sent to software router to be processed. The operating system acts as an "oracle" that always performs the "correct action," just in the sense of hardware/ software co-design ideas.

Of course, packets passing through software are processed more slowly than packets switched by the hardware engine. It does not have to indicate a serious problem if the portion of the software processed traffic related to the total traffic is small enough. On one extreme, we can achieve correct behaviour of the router sending all traffic to software processing. Surely, this is not what we intend. The goal is to accelerate as much traffic as possible.

Maximum performance would be possible only if all traffic was switched by hardware. Nevertheless, following kinds of reasons can prevent us to reach this goal.

- We can have "design reasons" (dictated by the economy of the development) not to process some types of packets in hardware. E.g., IPv6 Routing Option requests swapping two IPv6 addresses in the output editor. This operation is not suitable to perform in FPGAs as 128-bit string swapping would consume too large area of the chip or inadequate time, depending on the design. On the opposite, packets with Routing Option are very rare in the traffic.

  This category also contains handling erroneous packets. The easiest way to create an appropriate response (an ICMP message) is to send the packets to the operating system.

- We can use the *SW* action as a part of the design. For example, we use it to initiate the ARP protocol in the operating system in case that the ARP record for a destination is not known. This technique will be used later in this chapter.

- Resigning from hardware processing can be helpful in case of "runtime problems" as the last resort. In the process of lookup structure compilation, running out of available memory may occur or length of lookup branch may exceed an allocation block. We will discuss this possibility in Section 5.8—we will see that the method of combining routing, ARP, and filters is adaptable to various memory sizes.

4.2 Routing and ARP

While a software router usually searches for the longest matching routing entry in order to obtain a next hop and then it resolves its MAC address, the hardware engine has to solve both steps at once.

We will consider managing *forwarded* traffic only, ignoring packets destined to the host computer itself. Delivering packets to the host computer is based on the same principles, it differs in its inputs only—input packet filter has to be applied. We will discuss host-destined traffic in Section 5.7.

Combining routing and ARP tables (and even maintaining them together) is not a new idea. Open-source BSD clones (FreeBSD, NetBSD) use a single table to store both routing and ARP information. Nowadays, FreeBSD designers tend to split the tables, saying that handling the tables separately is easier and avoids unnecessary redundancies [Rizzo, 2004a].

The routing-ARP table resulting from this process is intended to be recomputed whenever either routing table or ARP table changes in order to keep the hardware lookup structure up-to-date.

In Section 4.2.1, we discuss how to obtain ARP records in the operating system. Moreover, we show that immediate recomputing of the routing-ARP table is not necessary in case of ARP record removal. In Section 4.2.2, we define the routing-ARP table formally. Section 4.2.3 describes the method to compute the routing-ARP table and demonstrates its correctness and properties of the method. In Section 4.2.5, we give the algorithm computing longest matching prefix representation of the routing-ARP table directly. Complexity of the resulting structure is given in Section 4.2.6.

4.2.1 Obtaining ARP Records

The routing table changes only when reconfigured by hand or by a routing protocol. This is not the case of ARP table. ARP table is quite dynamic, it holds records that have been used at most some time ago. This section discusses how to keep behaviour of the routing-ARP table equivalent to the routing table and ARP table while not changing the behaviour of the operating system itself. In the previous section, we stated that the routing-ARP table must be recomputed whenever either routing table or ARP table changes. We also demonstrate that this requirement is too strict and it can be softened. We show that in case of ARP record removal, recomputing the whole structure immediately is not necessary.

Records in the ARP cache are kept for a specified amount of time, typically from five minutes to several hours. In order to minimise the recomputations of the lookup structure, the initial idea was to test the whole network and to build the ARP cache as complete as possible[21]. Although this would be feasible in small networks, it is not scalable very well and it is unclean how to handle validity of records in such table. Moreover, it would require introducing a mechanism to learn ARP records into the operating system. It is not very suitable as one of the most important design principles is to keep the behaviour of the software router unchanged[22] whenever possible, so this approach was definitely rejected.

Instead of pro-active building the ARP cache, we use current content of the table, marking currently unresolved entries to be processed by software. The lookup structure must be updated when an ARP cache entry is added or modified. If a MAC address of the next hop of an incoming packet may be resolved, the packet

[21] Theoretically, this can be achieved trying to `ping` all machines in the local network.

[22] Both in sense of not changing the kernel and not changing the standard behaviour of the operating system.

is forwarded by hardware. Otherwise the packet is sent to the operating system. Software emits the ARP query to learn the MAC address of the next hop and enriches the ARP cache with the response. As the content of the cache changed, the lookup structure must be recomputed. Finally, when the lookup structure propagates into hardware, incoming packets forwarded to the previously added next hop will be handled by hardware.

A delicate problem arises with removing entries from the ARP cache. To keep the principle of equivalence of hardware and software behaviours, we should recompute the lookup structure when an entry from the ARP cache is deleted. Nevertheless, we have a reason against it, it is desirable to keep the update frequency of the hardware lookup structure as small as possible.

Let us study "observable" effects of entry removal in order to show that we may be reluctant removing the entries. The first reason to purge ARP entries is keeping the cache reasonably small, as pointed out by Malkin [Malkin, 1995] (the original RFC [Plummer, 1982] only addresses the problem stating this issue needs more thought). When a network node becomes inactive, no reason exists to keep an ARP record for it.

A MAC address belonging to an IP address may change (this is usually called "ARP moved", for an implementation, see, e.g., FreeBSD [FreeBSD, 2005] kernel code[23]). This may happen for example on a DHCP WiFi network. If the connection is initiated by the newcomer, the ARP cache of the receiving station will be corrected by the new record, so changing the entry causes lookup structure recomputation. While this behaviour seems very logical and conforming the principles of ARP, the author was not able to find any authoritative source describing or even standardising it. Custom and implementations seem to be the only source. However, some implementations do not behave this way. IP stack in the Sun Solaris operating system ignores ARP cache changes that should be caused by ARP move and waits until the entry expires[24]. Even though we have not verified this behaviour, we conclude that details of changing ARP records related to an IP address may be strongly implementation dependent. Anyway, it has no effect on recomputing the table: adding or changing a record must trigger routing-ARP table recomputation.

To sum up, we may omit updating the lookup structure in case of ARP cache entry deletion in case that updates stimulated by other causes are reasonably frequent (we can use a timer restarted by each update). Although we slightly violate the "hardware and software equivalence principle," lazy removing of the entries just lengthens the timeout. It may be suitable to use a timer that causes recomputation of the lookup structures in order of seconds from the last recomputation even if no direct reason for recomputation occurred. It ensures that the prolongation

[23] `netinet/if_ether.c`

[24] `http://supportforum.sun.com/network/index.php?t=msg&th=247&start=0&rid=0`

of the ARP table is kept “an order of magnitude” smaller than the normal ARP record timeout value.

4.2.2 RA Table Definition

Having current contents of routing and ARP tables as the input, the task is to compute the table that combines routing and ARP into the only lookup operation. We call the resulting structure a routing-ARP table:

$$RA\colon IP \to (Interfaces \times MAC) \cup \{SW\}$$

The RA table returns either the interface and MAC address of the next hop where the packet should be sent or it indicates that the packet must be processed by software.

Definition 4.1 RA table
Routing-ARP table RA is an ordered list of rules. We will use RA_i for *routing-ARP rules* (for $1 \leq i \leq \mathrm{Size}(RA)$) and $[RA_i]$ for IP addresses affected by the rule. Analogically with routing syntax, $\mathrm{NH}_{\mathrm{Int}}(RA_i)$ and $\mathrm{NH}_{\mathrm{MAC}}(RA_i)$ are the *interface* to reach the next hop and the *hardware address of the next hop*, respectively.

Let $\mathrm{Len}(RA_i)$ be the *length of the prefix* in the rule; RA records with length zero are called *default RA rules* and records with the length of the IP address (32 for IPv4 and 128 for IPv6) are called *full-length RA records*.

To obtain a *result* for a destination address $p \in IP$, denoted $RA(p)$, we find RA_i for the smallest i such that $p \in [RA_i]$.

4.2.3 Computing First-Match RA Tables

Creating the RA table out of routing and ARP tables in first-match representations is quite straightforward. Traversing the routing table, ARP entries are “injected” inside, depending on whether the route is direct or indirect. In case of locally connected network, all resolved ARP entries are added. Finally, the traffic destined to the rest of the network with currently unknown ARP records is delivered to the operating system. For indirect routes, the MAC address of the next hop is inserted into the table directly instead of its IP address if the MAC address is known. Otherwise, the route must be processed by software.

Algorithm 4.1 is a pseudocode for combining routing table represented as a list (which is equivalent to traversing a trie in non-increasing prefix length order) and ARP table (formally also first match, but it can be represented in an arbitrary

```
$RA = \emptyset$
$l = 1$
for $i = 1$ to Size($R$) do
    if $R_i$ is direct
        then
            /* go through all ARP records in this local subnet: */
            for each $j$ such that $1 \le j \le$ Size($A$) and $[A_j] \subseteq [R_i]$ do
                $[RA_l] = [A_j]$
                $RA_l = (\mathrm{NH}_{\mathrm{Int}}(R_i), \mathrm{NH}_{\mathrm{MAC}}(A_j))$
                $l = l + 1$
            done
            /* the rest must get resolved in software */
            $[RA_l] = [R_i]$
            $RA_l = SW$
            $l = l + 1$
        else /* $R_i$ is indirect */
            if exists $k$ such that $\mathrm{NH}_{\mathrm{IP}}(R_i) = [A_k]$
                then
                    $[RA_l] = [R_i]$
                    $RA_l = (\mathrm{NH}_{\mathrm{Int}}(R_i), \mathrm{NH}_{\mathrm{MAC}}(A_k))$
                    $l = l + 1$
                else
                    $[RA_l] = [R_i]$
                    $RA_l = SW$
                    $l = l + 1$
            fi
    fi
done
if $l = 1 \vee [RA_{l-1}] \neq IP$ /* $\vee$ requires lazy evaluation */
    $[RA_l] = IP$ /* ensure that $RA$ contains a default route */
    $RA_l = SW$
fi
```

Algorithm 4.1 Combining routing and ARP into RA table

manner, say a list—ARP records are not ordered) into the RA table represented as a list.

We have to show that the RA table computed by this algorithm behaves the same way as routing and ARP tables. It means that the RA lookup result is the same as ordinary software processing where lookups in routing table and resolving ARP are performed separately. Sending the packet to software is also a correct result—in that case the packet is processed by the original routing and ARP tables.

First, we show that Algorithm 4.1 performs "useful work," i.e., that it does not result into sending all packets to the operating system. For each direct route, routing-ARP records are created for all destinations if the appropriate ARP record is known. If the ARP record of a gateway is known it is used to create a routing-ARP record for the indirect route.

Lemma 4.2
RA table produced by Algorithm 4.1 is total (i.e., it is defined for each destination address $p \in IP$).

Proof
To ensure that the RA table is total, it is sufficient that it contains the default RA record.

If the routing table contains the default route then the default route must be the last record in the table and the default RA record is inserted into the RA table. At the end, the algorithm tests presence of the default route in the table. If the default route was not present the algorithm adds final the RA record destined in software. □

We have shown that the result of RA lookup is defined. In the following theorem, we prove that the result is correct.

Theorem 4.3 Correctness of RA compilation
For each destination address $p \in IP$, routing-ARP table contains a result that is either SW or it is the same as applying routing and then ARP on the destination address, i.e.,

$$\begin{aligned} \mathrm{NH}_{\mathrm{Int}}(RA(p)) &= \mathrm{NH}_{\mathrm{Int}}(R(p)) \\ \mathrm{NH}_{\mathrm{MAC}}(RA(p)) &= \mathrm{NH}_{\mathrm{MAC}}(A(\mathrm{NH}_{\mathrm{IP}}(R(p)))). \end{aligned}$$

Proof
Let us have a destination address $p \in IP$. As the RA table is total, the smallest index l exists such that $p \in [RA_l]$.

If $RA_l = SW$ then the destination address is handled by the original routing table and ARP cache in software, thus correctly.

Let us suppose the result is an interface $\mathrm{NH}_{\mathrm{Int}}(RA_l)$ together with a MAC address $\mathrm{NH}_{\mathrm{MAC}}(RA_l)$. In that case, the RA rule RA_l has been created either in lines 8–10 or lines 19–21 of the algorithm. We will study both cases.

Let us suppose that the rule RA_l was created in lines 8–10. Such an entry has a single prefix of full length in $[RA_l]$ (it originates from the ARP table that contains full addresses). We will show that the RA_l rule was created from a route R_i where i is the smallest index such that $p \in [R_i]$. Suppose the opposite. Then an index j such that $1 \leq j < i$ must exist such that $p \in [R_j]$. Length of prefix of R_i cannot be equal to $\text{Len}(R_j)$ as prefixes of equal lengths are disjoint. Therefore $\text{Len}(R_j) > \text{Len}(R_i)$ due to the ordering of R. As each routing record produces at least one routing-ARP record with the same address space (the final software entry for direct routes and/or the indirect route), the algorithm must have produced an entry RA_k such that $1 \leq k < l$ and $p \in [RA_k]$. This is not possible as we chose the RA_l as the first matching entry for destination address p. Hence $\text{NH}_{\text{Int}}(RA(p)) = \text{NH}_{\text{Int}}(R(p))$.

As $\text{NH}_{\text{IP}}(R(p)) = p$ for direct routes, a unique ARP entry A_j exists such that $\text{NH}_{\text{MAC}}(RA_l) = \text{NH}_{\text{MAC}}(A_j)$.

The remaining case is that the rule RA_l was created in lines 19–21. We will show that the rule originates from the first R_i for that $p \in [R_i]$. Suppose the opposite, precisely, that a routing record such that $p \in [R_j]$ exists for $1 \leq j < i$. Then again, $\text{Len}(R_j) > \text{Len}(R_i)$, moreover each routing record produces at least one RA entry, therefore an RA record RA_k must exist such that $1 \leq k < l$ and $p \in [RA_k]$. That is not possible due to the choice of RA_l as the first matching record.

To sum up, $\text{NH}_{\text{Int}}(RA_l) = \text{NH}_{\text{Int}}(R_i)$ and $\text{NH}_{\text{MAC}}(RA_l) = \text{NH}_{\text{MAC}}(A(\text{NH}_{\text{IP}}(R_i)))$.

□

We have obtained a routing-ARP table that is behaviourally equivalent to the original routing and ARP tables. The RA table is first match but it can contain redundancies. It is advantageous to have a trie form of the table (i.e., an equivalent longest matching prefix representation) in order to obtain a concise representation. We will use trie representation to combine the structure with packet filters in Chapter 5.

If we are able to sort RA entries in non-increasing way and ensure that prefixes of equal lengths are disjoint we would obtain a table that can be converted into a trie form. To reach this goal, let us study the ordering and relations of rules produced by the algorithm and potential redundancies in the table that have to be discarded.

Lemma 4.4 RA compilation sort order

Algorithm 4.1 sorts the resulting RA table in non-increasing prefix lengths with exception of prefixes of full lengths, i.e., for $1 \leq k < l \leq \text{Size}(RA)$ the following condition holds: $\text{Len}(RA_k) \geq \text{Len}(RA_l)$ or RA_l has full length.

Moreover, prefixes with equal lengths with exception of full-length prefixes are disjoint.

Proof

The RA records are copied in the same order as the original routes with the exception of resolved local ARPs that can produce full addresses.

Each routing record produces just one RA prefix of the same length and optionally a number of full-length prefixes. Note that the final default RA record

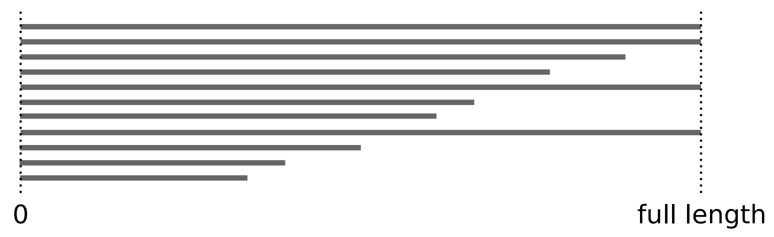


Figure 4.1 Scheme of prefix length structure created by Algorithm 4.1

destined in software is added only if no default record was present in the table (see Lemma 4.2). □

The structure of lengths of entries is shown in Figure 4.1. As we see from the preceding lemma, the only issue that remains to be solved is the structure of full-length prefixes. The full-length prefix for a particular destination address may be repeated across the table several times. In that case the first occurrence is of interest as all the following ones are unreachable and may be omitted.

Lemma 4.5 Deleting redundant RA entries
Let us have $1 \leq k < l \leq \text{Size}(RA)$ such that $[RA_k] = [RA_l]$. Let RA' be the table RA with row l omitted ($RA'_i = RA_i$ for $1 \leq i < l$ and $RA'_i = RA_{i+1}$ for $l \leq i < \text{Size}(RA)$). Then $RA(p) = RA'(p)$ for each destination address $p \in IP$.

Proof
Obviously, the row RA_l is unreachable in the original table. □

Full-length prefixes are mixed into prefixes of other lengths. In order to prepare the table to be converted in longest prefix semantics it would be more suitable to move the prefixes to the beginning of the table. We have to make sure that this operation does not change the meaning of the table. To ensure this, it is enough that the address space affected by the prefix is disjoint with all the preceding rules (i.e., rules of an arbitrary length). Blocks of disjoint records can be freely re-ordered.

Thanks to the preceding lemma, we have to be interested in the *first* occurrence of the full-length prefix for an address only.

The meaning of the following result is that the first occurrence of a full length entry in the RA table is disjoint with *all* its predecessors. It can also be understood from an operational point of view that the first occurrence of a full length entry for an address is *reachable* in the routing-ARP table.

Lemma 4.6 Sorting full-length RA records
Let RA_l be the first occurrence of full-length RA record for a given destination address, precisely, let RA_l be a full-length RA record satisfying that for each full-

length record RA_m such that $[RA_l] = [RA_m]$ we have $l \leq m$. Then for each k such that $1 \leq k < l$ condition $[RA_k] \cap [RA_l] = \emptyset$ holds.

Proof
Let $p \in [RA_l]$ satisfying the prerequisites of the Lemma. We want to show that no index k exists such that $1 \leq k < l$ and $p \in [RA_k]$. Let us suppose the opposite, i.e., that a routing-ARP entry RA_k exists such that $p \in [RA_k]$ and $1 \leq k < l$ (saying nothing about its length). We will study all possibilities how such an entry could have been created.

Let R_j be the routing record from which RA_l was created and R_i the record that produced RA_k. First, we note that $p \in [R_j]$ and $p \in [R_i]$. If it was not so, the RA entries would not contain p (the algorithm copies the address space and/or takes a full-length entry out of the address space of a direct route). Moreover, $i \leq j$, otherwise the order of the produced routing-ARP entries would not be preserved. (Note that equality is possible, a routing rule can produce several routing-ARP entries.)

The route R_j can be either indirect or direct. We will study both cases.

- Let R_j be indirect[25] first. Then R_j must have full length, otherwise it would not produce a full length entry RA_l. As $i \leq j$ the route R_i must have full length due to the ordering of the routing table. If $i < j$ then $[R_i]$ and $[R_j]$ would be disjoint. This is not possible as both contain p. Hence $i = j$. As indirect route produces just one routing-ARP entry, we cannot have $k < l$. Therefore R_j cannot be indirect.

- Let R_j be direct. In that case the route R_i must be direct or full-length due to the requirement 3 on page 36.

  - Let R_i be direct. As R_j is direct, the record RA_l must have been created either in lines 8–10 or lines 13–15 of the Algorithm 4.1.

    ⋆ If RA_l was created in lines 8–10 then an ARP record for p must exist. It would be used also to expand the route R_i, hence RA_l would not be the first full-length record for p.

    ⋆ If RA_l was created in lines 13–15 then R_j would have full length. As we have $i \leq j$ then route R_i would have full length, too.

      We show that $i = j$. If $i < j$ then $[R_i]$ and $[R_j]$ would have to be disjoint. This is not possible as both contain p. Hence $i = j$.

[25] This is a theoretical option as it makes no sense to have subnets of full lengths normally. Anyway, e.g., Linux `route` command allows to configure them, therefore we have to allow such pathological cases in the model.

It is possible that the route R_i created two (full-length) entries for p. In that case RA_l would not be the first full-length routing-ARP entry containing p.

– The remaining case is that R_i has full length. Then R_i must have created a full-length record RA_k such that $p \in [RA_k]$. Then RA_l is not the first full-length record handling p, which is not possible.

The record RA_k could not be created in any of the cases. □

To prove the preceding lemma, we supposed that all subnets of local networks are direct or have full length (see requirement 3 on page 3). If it was not so, an unreachable full length entry might be produced by the algorithm. Consider a fragment of a routing table as an example:

```
1.2.3.0/24 -> indirect route
1.2.0.0/16 -> direct route
```

Let us suppose that an ARP record exists for the `1.2.3.4` destination address and the indirect route has just been added. The ARP record will have been occupying the ARP cache for several minutes. The algorithm would produce

```
1.2.3.0/24 -> some gateway and/or SW
1.2.3.4/32 -> resolved ARP in the local network
1.2.0.0/16 -> some gateway and/or SW
```

The second record is unreachable and cannot be moved to the beginning of the table.

4.2.4 RA Table Properties Summary

To sum previous results up, we have observed following properties of the routing-ARP table produced by the Algorithm 4.1:

1. Only the first occurrence of an address space is of interest, precisely if the algorithm creates an entry RA_l and an entry RA_k such that $[RA_k] = [RA_l]$ and $k < l$ has been created before, we do not have to store RA_l into the RA table according to Lemma 4.5.

2. Full-length entries may be pushed to the beginning of the table thanks to Lemma 4.6. This way we obtain a table sorted in non-increasing prefix lengths.

This algorithm is the same as Algorithm 4.1 with the following modifications.

- The resulting RA table is kept in a trie structure. (Hence it is sorted on-the-fly in non-increasing way and all records are unique.)
- Let us understand assignments that create rules RA_l in the following way:
  - Assignment to $[RA_l]$ is creating a path in the trie denoting the prefix of the RA_l rule.
  - Assigning the output to the rule (e.g., $RA_l = (\mathrm{NH}_{\mathrm{Int}}(R_i), \mathrm{NH}_{\mathrm{MAC}}(A_j)))$ means *assign the output only if it was empty previously*.

Algorithm 4.2 Combining routing and ARP into RA table expressed as trie

4.2.5 Longest Prefix Representation of RA Table

Instead of post-processing the table, we may change the order of the rules immediately during the run of the algorithm with a small change of semantics of assignments. The final version of the computation is shown in Algorithm 4.2.

With those changes, the algorithm constructs a trie representation of the RA table which is equivalent to the original output list. Only the first rule for a given address prefix is recorded and full-length entries may be harmlessly "moved to the beginning" as we have shown above. Again, first match representation may be obtained from the trie traversing it in post-order manner. Also note that rules sharing common prefix length are disjoint.

The RA table computed out of examples shown in Figure 3.1 and Figure 3.3 is depicted in Figure 4.2.

```
147.251.54.1/32 -> eth0, 00:E0:81:27:DF:7B
147.251.54.10/32 -> eth0, 00:20:ED:5E:6D:98
147.251.54.0/24 -> SW
0.0.0.0/0 -> eth0, 00:E0:81:27:DF:7B
```

Figure 4.2 RA table computed out of R in Fig. 3.1 and A in Fig. 3.3

4.2.6 Complexity of the RA Table

Space needed to store the RA table is limited from above by $O(\mathrm{Size}(R) + \mathrm{Size}(A))$. The RA table contains a copy of the routing table. Moreover, some networks can be expanded into several records originating from the ARP table. Each ARP record can be inserted at most once as we can see in the Algorithm 4.2.

Time to search the RA table in software is completely the same as for routing tables kept in trie structures, i.e., $O(w)$ where w is bit length of an address. At this point, we shall note that this part of the structure is kept in CAM and searched in constant time. We will discuss this topic in detail in Section 5.6 where packet filter representation is also considered.

5 Routing, ARP, and Filtering Combined

No, my good lord, but, as you did command,
I did repel his letters and denied
His access to me.
– William Shakespeare, Hamlet, Act 2, Scene 1

In Chapter 4, we have combined routing and link layer addressing. In order to obtain a single lookup structure covering routing, ARP, and packet filter, we have to add packet filter setting and convert the structure to the target architecture—a first match CAM and comparison instructions.

First, we define a structure called routing-ARP-filtering (RAF) table. Its type describes how packet processing actions can be combined. Theoretically, the RAF table can be computed as a Cartesian product of the RA table and the packet filter. The result is behaviourally correct, alas, it cannot be rewritten into the target architecture reasonably. We demonstrate that such approach is completely unable to utilise advantages of content addressable memories. More sophisticated method is therefore necessary.

The main problem of the naive approach is that it is not able to rearrange RAF records produced into first match structure. We need a way to re-sort the records into the required form without changing their semantics. We introduce representation of a packet filter as a Filtering Decision Diagram (FDD) which is basically a special type of a binary decision diagram with semantical relationship among nodes. FDD nodes test terms of filtering language. It differs from traditional binary decision diagram methods where a single bit is the unit of processing. Moreover, we introduce encoding filtering rule priorities (i.e., order of the filter) directly to the FDD, without the need to compute partitioning of packet space explicitly.

To obtain a single lookup structure, the routing-ARP table can be combined with FDD representation of packet filters. We moreover distribute relevant filters to relevant portions of the destination address space. It avoids testing filters that would not match anyway for a given destination address. Finally, an algorithm to convert the "upper part" of the RAF representation into first match structure that can be interpreted by CAM is shown.

This chapter is organised as follows. In Section 5.1, we define the RAF table. Section 5.2 presents the naive method of RAF table computation. Section 5.3 defines filtering decision diagrams, Section 5.4 describes how FDD representation of filters can be combined with RA tables. In Section 5.5, we demonstrate how the resulting structure can be employed in the target architecture. We give a complexity estimate in Section 5.6, Section 5.7 describes how traffic destined to the host computer is handled. Finally, Section 5.8 points out properties of the method allowing to use it reasonably on limited hardware resources.

In the major part of this chapter, we discuss *forwarded traffic processing* only unless said otherwise.

5.1 RAF Table

Routing-ARP-filtering (RAF) table is a structure combining routing-ARP table into a single structure with packet filters. Its type originates from possible results of the classification. The result is one of the following:

- The RA result is the next hop interface and MAC address, the filtering rule is accepting, and the filtering action can be performed by the accelerator. Then the RAF result is the interface and MAC address together with all action modifiers accompanying the filtering rule.

- The filtering action avoids sending the packet to the next hop (e.g., dropping the packet) and the action can be performed by the accelerator. Then, just the filtering action shall be performed.

- Otherwise, the packet must be sent to the operating system, so the remaining possibility is the *SW* action.

Let us express the possibilities formally together with syntactical constructs to handle RAF tables.

Definition 5.1 RAF table
Routing-ARP-filtering (RAF) table is a total function

$$RAF\colon PktInfo \to (Interfaces \times MAC \times 2^{ActionModifiers}) \cup Actions \cup \{SW\}$$

where *PktInfo* is a packet information (described in Section 3.3). *Interfaces* is a set of output interfaces, *MAC* is a set of MAC addresses. *ActionModifiers* is a set of action modifiers we have described in Section 3.3.4. In this case, the packet is forwarded and some additional processing (that can be performed by the accelerator) is required. In case the packet is not forwarded, *Actions* is a set of possible filtering actions (we recall that $Actions = BasicActions \times 2^{ActionModifiers}$). Finally, *SW* is a special action requiring the packet to be processed by the operating system IP stack.

Similarly with RA table and packet filter syntax, we use RAF_i for $1 \leq i \leq \text{Size}(RAF)$ to denote RAF records. On the "input side," $\langle RAF_i \rangle \in PktInfo$ is the packet information that rule RAF_i matches and $[RAF_i] \in IP$ is the matching set of destination addresses. A rule such that $\langle RAF_i \rangle = PktInfo$ is called a default RAF rule.

The result of applying RAF table on packet information $p \in PktInfo$ is the first matching RAF record, formally $RAF(p) = RAF_i$ for the smallest index i such that $p \in \langle RAF_i \rangle$ for $1 \leq i \leq \text{Size}(RAF)$.

If the output of the RAF_i rule is an output interface with a MAC address, we use $\text{NH}_{\text{Int}}(RAF_i)$ and $\text{NH}_{\text{MAC}}(RAF_i)$ to obtain them. By $\text{Action}(RAF_i)$ we denote the set of actions related to the rule. Let us recall that although the *Actions* set is internally structured, we use the "$\in$" relation for all of its components (see Section 3.3.4 for details).

We require the *RAF* table to be total, an action must be taken for each packet. The table can be easily made total adding a default rule, say destined in software.

5.2 Naive Approach

The first idea how to combine routing, ARP, and packet filtering into a single operation is partly inspired by expansion/compression approach [Crescenzi et al., 1999a]. We tried to construct the table as a Cartesian product of the RA table and the packet filter and to optimise it later. We briefly describe the method. Finally, we show that the method does not produce a structure that can be re-written efficiently into a lookup structure suitable for machines combining associative and conventional memories.

5.2.1 The Method

We build a Cartesian product of the RA table and the packet filter as shown in the scheme in Algorithm 5.1. The function create_raf_entry(RA_i, F_j) produces an RAF entry of format "$[RA_i]$ and $\langle F_j \rangle$", i.e., the routing prefix and the filtering rule in conjunction.

We suppose that we have a list of actions that can be performed by the accelerator, see Section 3.3.4 for details. The outcome of the RAF rule depends on actions prescribed by RA_i and F_j:

- if following conditions hold:

```
$l = 1$
for $i = 1$ to Size($RA$) do
    for $j = 1$ to Size($F$) do
        $RAF_l$ = create_raf_entry($RA_i, F_j$)
        $l = l + 1$
    done
done
```

Algorithm 5.1 Combining RA and packet filter naively

- the result of RA_i is not SW,
- the action in the F_j entry is accepting, and
- the action in the F_j can be performed by hardware

then the resulting interface is $\mathrm{NH}_{\mathrm{Int}}(RA_i)$, MAC address $\mathrm{NH}_{\mathrm{MAC}}(RA_i)$, and action modifiers from the F_j if present,

- if the F_j action is not accepting but can be performed by hardware, the action in the RAF rule is the action (including action modifiers) from F_j,
- the action is SW otherwise (in this case, the action cannot be performed by hardware or the packet is destined to the host machine).

Note that even if the RA entry results in SW and the filter drops the packet with an action that can be performed by hardware, the packet will be dropped by the accelerator and it will not be transferred through the system bus.

Various optimisations can decrease the number of records produced by the algorithm. We can omit entries that are not satisfiable because of disjoint requirements on destination addresses originating from the route and from the filter. We can remove unreachable entries, records resulting into identical actions can be merged in some cases. Alas, no optimisations can prevent the situation described in the following Section 5.2.2.

5.2.2 Table Degeneration

The resulting RAF structure must be re-written into a structure that can be interpreted by the lookup processor.

First, let us sum up basic properties of the lookup engine design relevant to the relationship of the RAF structure and the resulting LUP nanoprogram. (Compare

```
dport 5 sport 1 A1                0.0.0.0/0 dport 5 sport 1 A1
dport 5         A2                0.0.0.0/0 dport 5         A2
dport 6 sport 2 A3                0.0.0.0/0 dport 6 sport 2 A3
        sport 2 A4                0.0.0.0/0         sport 2 A4
                A5                0.0.0.0/0                 A5
```

Figure 5.1 A packet filter (left) and the resulting RAF table (right)

with the overview of the target architecture and its abstract properties in Chapter 1.)

- Width of the CAM is not sufficient to perform match of the full width, therefore some parts of *PktInfo* cannot be matched in CAM and must be handled by comparison instructions.

- Parts ("columns") of *PktInfo* that are matched in CAM are chosen arbitrarily but globally. (We can choose what fields shall be matched in CAM but we have to do it for the whole structure.)

- When a packet matches a CAM line, comparison instructions belonging to the line have to resolve the search completely. No possibility exists to match the pattern against remaining CAM lines again. In other words, backtracking has to be avoided.

The method to compute the RAF table described above produces a first match rule list that could be used if it fitted into CAM width. The question is whether we can give a general method to find rules what columns shall be moved to the instructions and how to rewrite the table into such form.

Let us consider the following example. We have an RA table containing just the default route (i.e., a route for `0.0.0.0/0`) pointing to a gateway. A filter is shown in the left part of Figure 5.1. The filter leads to various actions `A1`, ..., `A5`, so the rules cannot be re-sorted in any way. Note that all the filtering rules are reachable. In short, the filter does not contain any pathology that could be suspected to cause unwanted effects in the resulting RAF table.

The RAF table computed according to the scheme in Algorithm 5.1 is shown in the right part of the figure. The question is how to rewrite it into CAM and SRAM instructions. Based on the properties of lookup engine design, let us suppose that we can match either source and/or destination port in CAM but not both at the same time. Hence, we have to aggregate entries that become identical in CAM and prepare the instructions that resolve them fully.

First, let us suppose that destination port is matched by CAM and source port by comparison instructions in SRAM. Then we obviously have to aggregate the identical lines:

```
[A1,A2] 0.0.0.0/0 dport 5 -> resolve between A1, A2 in SRAM
[A3]    0.0.0.0/0 dport 6 -> if sport 2 then A3
[A4,A5] 0.0.0.0/0 dport * -> if sport 2 then A4 else A5
```

We use brackets to denote origins of the lines. The content of CAM follows. The text following the "`->`" sign describes the structure of the SRAM instructions. Symbol "`*`" stands for don't cares.

As we can see, packet with destination port 6 and source port 1 would be caught by line `[A3]`. It should match the `[A4,A5]` instead. It is caused by the fact that the SRAM resolution of the `[A3]` line is not total (in the sense that it does not have to decide the search fully). To correct that, we could compact lines `A3`–`A5` and to test the destination port again by instructions, completely wasting CAM space. The resulting structure would be:

```
[A1,A2] 0.0.0.0/0 dport 5 -> resolve between A1, A2 in SRAM
[A3-A5] 0.0.0.0/0 dport * -> resolve A3, A4, and A5 in SRAM
```

Another possibility is to preserve the `[A3]` line and to push resolving the rest of the filter there.

To show that this problem is not caused by wrong choice of fields that shall be matched in CAM, we will now match source port there. Because of the don't-care entry on the second RAF entry, we obtain

```
[A1]    0.0.0.0/0 sport 1 -> if dport 5 then A1
[A2-A5] 0.0.0.0/0 sport * -> resolve A2-A5
```

Even this representation requires backtracking; packets with source port 1 and destination port other than 5 would be misled to the `[A1]` instruction set. The way to represent this structure correctly is not to resolve the complete ruleset by the instructions, using the only CAM row. This is caused by entries unspecified in the filter crossed thorough its columns. We call this situation "crossed don't-cares." Crossed don't-cares can even cause that all entries have to be moved from CAM. Moreover, crossed fields do not have to be localised in consecutive entries. Packet filters tend to specify only several fields in a single rule, so we can expect that the density of crossed don't-cares is high.

We could also change the order of rules (hoping it would produce better results) by swapping how the cycles in Algorithm 5.1 are nested. Note that it would not change anything in our example.

5.3 Representing Filtering Rules as Decision Diagrams

The method of combining RA table with filtering rules we have described in Section 5.2 seemed simple and obvious, but it turned out to be completely unusable in the target hardware architecture.

The problem of crossed don't-cares initiated a search for a method that allows re-sorting the records in more or less arbitrary manner. In an ideal case, such resorting requirements should be satisfied "behind the scenes," guaranteeing equivalent representations without explicit interaction.

The structure suggested by Vojtěch Řehák to study for this task was a binary decision diagram. We developed a special type of this structure—filtering decision diagram (FDD)—that reflects natural properties of packet filters (like typical types of tests and rule priorities) on one hand and requirements of target architecture on the other one, i.e., possibility to rewrite such structure into a hardware machine employing content addressable memory and comparison instructions.

In the following Section 5.3.1, we define the filtering decision diagram. *Variables* of the filtering decision diagram (FDD) are directly terms of the filtering language. Sections 5.3.2 and 5.3.3 describe basic procedures to manipulate FDDs. The procedure reflects *semantical relations among FDD variables*. Conversion of a simple rule is presented in Section 5.3.4. We introduce *encoding rule priorities directly* in the FDD. Section 5.3.5 describes converting the whole rule set to an FDD. Section 5.3.6 summarises notes on practical implementation.

5.3.1 Filtering Decision Diagrams

After recalling classical definitions of binary decision diagrams and their special types, we define filtering decision diagrams in this section.

Definition 5.2 (Multi-terminal) binary decision diagram
Binary Decision Diagram (BDD) [Bryant, 1986] is a rooted directed graph (V, E) with two types of vertices. Each *terminal* vertex is labelled 0 or 1. A *nonterminal* vertex v is labelled with a Boolean variable $\mathrm{var}(v)$ and it has two successors, $\mathrm{low}(v)$ corresponding the case when the variable is assigned 0, and $\mathrm{high}(v)$ for 1.

Multi-Terminal BDD (MTBDD) [Bahar et al., 1993][26] is defined the same way as BDD, only the terminal vertices are labelled with elements of a finite set.

[26] The structure is called Algebraic Decision Diagram (ADD) in that paper.

For an assignment of variables, the value of the function represented by the BDD is obtained by traversing a path from the root to a terminal node, choosing branches indicated by the values of the variables.

Definition 5.3 MTBDD—special types
A (MT)BDD is *ordered* if on all paths from the root to a terminal node the variables respect a given linear order.

A (MT)BDD is called *reduced* if it satisfies following conditions:

- (uniqueness) no two distinct nodes u and v have the same variable and low- and high-successor, i.e., $\mathrm{var}(u) = \mathrm{var}(v)$, $\mathrm{low}(u) = \mathrm{low}(v)$, $\mathrm{high}(u) = \mathrm{high}(v)$ implies $u = v$, and

- (non-redundancy) no variable node u has identical low- and high-successor, i.e., $\mathrm{low}(u) \neq \mathrm{high}(u)$.

We will define a structure called Filtering Decision Diagram (FDD) to represent packet filters. FDD is a special type of MTBDD. For the purpose of defining an FDD, we must first discuss its variables and terminals.

In Section 3.3.5, we defined a packet filter. We use the syntax described there to show the transformation of the filter to the diagram representation. It does not affect generality as the language contains all possible types of queries: exact match, prefix match, and range checks. Our approach can be therefore easily modified to include broader repertoire of tests. Let us recall that the filtering language consisted of rules containing conjunctions of filtering terms.

Definition 5.4 FDD variables
By *FDD variables* we understand the set of all possible filtering terms of the filtering language.

All variables testing a single field (i.e., `sif`, `dif`, `saddr`, `daddr`, `sport`, `dport`, `proto`, cf. Figure 3.6) is called a *class* of FDD variables. E.g., all tests of `dport` ranges belong to a single class which is distinguished from a class of `saddr` tests.

We distinguish three types of FDD variables[27]:

[27] We must be aware of the fact that the classification is extremely fuzzy and serves mostly to show examples. Real-world filtering languages often support constructs like `proto 0-50` as protocols are also numbered.

1. exact match checks for protocols and interfaces, e.g., `proto tcp`,
2. prefix match checks for addresses, e.g., `saddr 147.251.54.0/24`,
3. range checks for ports, e.g., `dport 1024-65535`.[28]

Terminals of the diagram are *Actions* of the packet filter. Moreover, we need a special terminal symbol called *HSL* (which stands for "hic sunt leones"). It denotes the position of the filter that corresponds to "the remaining filter rules," as we will see in Section 5.3.5. During the computation of the filter, it will be overwritten step-by-step by representations of subsequent rules.[29]

Definition 5.5 Filtering decision diagram
A Filtering Decision Diagram (FDD) is an MTBDD over FDD variables. Its terminals are from $Actions \cup \{HSL\}$.

We say that an FDD is *finished* if it does not contain the *HSL* terminal.

We use term *reduced* FDD (RFDD) in the same sense as for BDD.

An FDD is *ordered* (OFDD), if all paths of the FDD respect an order of variables $<$ (variables smaller in the $<$ relation appear higher in the FDD).

Motivation for ordering the FDD is twofold:

- it may make processing of FDDs faster (allowing to stop the search earlier in recursive procedures traversing the structure),
- it may help rewriting the structure into the format of the target architecture (i.e., first-match CAM where order of columns is prescribed).

We define the functions as order-independent as possible and the differences are commented. We consider several types of ordering.

1. Total order. Relation $<$ is a linear order over FDD variables.
2. Class order. We prescribe the order of classes of variables, e.g., we require that all `dport` tests precede `saddr` etc. Variables inside a class are incomparable.

[28] For handling range queries, it might be useful to allow variables of the form `dport >= 1024`. Nevertheless, we can use tests with mandatory lower and upper bounds as all the domains are finite. We prefer not to make the theory more complex than necessary.

[29] Although *HSL* is a terminal symbol from the decision diagram point of view, it can be understood as a non-terminal symbol in the process of filter composition.

3. No order at all, all the variables are incomparable. For the purposes of the algorithms we present, we define the < relation to never hold. It allows to write the algorithms in a uniform manner. (We do not call such FDD ordered.)

We will now define basic functions to handle FDDs. The functions are derived directly from standard BDD operations as described, e.g., in [Andersen, 1998a]. Notes on principles of efficient BDD implementation can be found in [Brace et al., 1991]. The FDD procedures have been implemented by Minaříková [Minaříková, 2005]; implementation details (e.g., node storage and memory handling) are also discussed there.

5.3.2 Creating and Testing FDD Nodes

Function `FDDCreate(`n, l, h`)` is a standard procedure for creating nodes in reduced BDD. It returns a node u with var(u) = n, low(u) = l, and high(u) = h. The nodes are stored in a hash table. When a suitable hashing scheme is employed, the complexity of this function is constant on average [Andersen, 1998a].

The function first tests if $l = h$. In that case, l is returned immediately in order to preserve non-redundancy. Otherwise, if the desired node is already present in the hash table, it is returned, if not, it is newly created. It ensures the uniqueness property. Using `FDDCreate(`n, l, h`)` in all cases when new nodes are created guarantees that the structure remains reduced; we have to take care of the order of variables only.

When total variable ordering is used, using `FDDCreate(`n, l, h`)` ensures that the FDD representation is canonical [Bryant, 1992]. When we relax such a strict requirement, we can obtain a pair of semantically equivalent structures that cannot be unified because of they are not equal syntactically, so their equivalence cannot be recognised. This behaviour is completely the same as for BDDs.

We define function `FDDIsTerminal(`n`)` that returns true if and only if the FDD rooted by node n is terminal.

5.3.3 Restriction

Given an FDD u (i.e., the root of an FDD) and an assignment of a variable j (e.g., `saddr 147.251.54.0/0`) to be either high or low, restriction `FDDRestrict(`u, j, v`)` computes an FDD under that assignment. Parameter v is the required value for that assignment (high or low).

Intuitively, the result of the restriction is the following: "for v = high, we suppose that variable j is satisfied, compute an FDD that does not test it again" and vice versa, i.e., if v = low then we compute an FDD that supposes that j is not satisfied and does not check it. The question we ask at each node is "based on

```
1  function FDDRestrict($u, j, v$)
2     if FDDIsTerminal($u$) then return $u$
3     fi
4     if $v$ = high then /* $j$ holds */
5        if $j \subseteq \mathrm{var}(u)$ then
6           return FDDRestrict(high($u$), $j, v$)
7        fi
8        if $j \cap \mathrm{var}(u) = \emptyset$ then
9           return FDDRestrict(low($u$), $j, v$)
10       fi
11    else /* $v$ is low, we know $j$ does not hold */
12       if $j \cup \mathrm{var}(u)$ = whole domain of variable var($u$) then
13          return FDDRestrict(high($u$), $j, v$)
14       fi
15       if $\mathrm{var}(u) \subseteq j$ then
16          return FDDRestrict(low($u$), $j, v$)
17       fi
18    fi
19    if $\mathrm{var}(u) < j$ then /* the variable is surely not present any more */
20       return $u$
21    else
22       return FDDCreate(var($u$),
23          FDDRestrict(high($u$), $j, v$), FDDRestrict(low($u$), $j, v$))
24    fi
25 end function
```

Algorithm 5.2 FDDRestrict(u, j, v)

the fact we know the result of test j, is the test in this node necessary?" We remove *all* tests in the FDD that are not necessary based on such knowledge. The pseudocode is shown in Algorithm 5.2.

The basic principle of computation is that we search for all nodes with the restricted variable and replace them with their low- or high-child depending on the variable evaluation.

In a standard BDD, no relationship among variables exists. This is not the case of FDDs. Knowledge about a variable may also allow restricting other variables belonging to the same class. E.g., if we know that `dport 1024-65535` condition holds, we may also reduce a condition `dport 0-25` as it is obviously false. This principle is expressed in lines 4–18 of the algorithm and Figure 5.2 where a case of interval comparison is shown. We call such a restriction *semantical*.

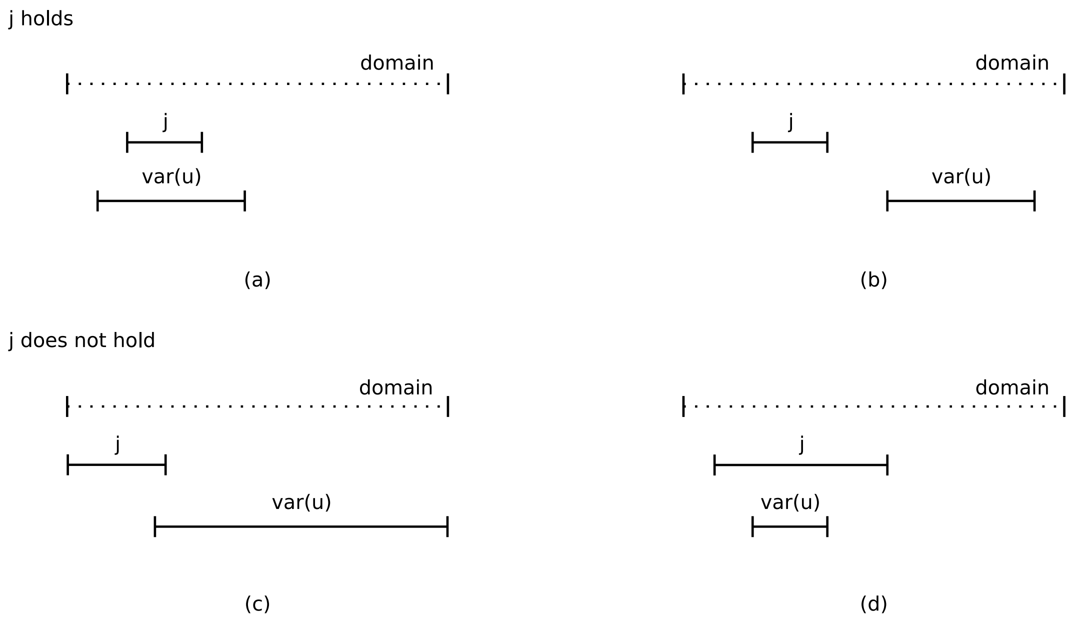


Figure 5.2 Principles of FDD restriction

Each FDD variable represents a set of packets matching the variable. Set relations and operations over FDD variables are performed over sets of matching packets.

Let us start with the case we require the variable j to hold (i.e., v = high). In that case, if condition $j \subseteq$ var(u) holds then testing var(u) is not necessary, e.g., for j = `dport 25-80` and var(u) = `dport 0-1023`. In that case, we can restrict the node u to high(u) (Figure 5.2 a).

If variables j and var(u) are disjoint, and we know that test j holds, we do not have to test var(u) as it would not hold anyway. We can restrict node u to low (Figure 5.2 b). In other cases, no restriction can be applied—we are not able to determine the relationship of the variables.

The other possibility is that we require v = low, i.e., we know that variable j is not satisfied. Then, tests on variables represented by j may be satisfied only in the complement of j. E.g., if j = `dport 0-1023` we can conclude that the `dport` test may hold for values `1024-65535`. Therefore, if var(u) covers the whole complement of j (which is expressed as "$j \cup$ var(u) = whole domain of variable var(u)" in the algorithm), then we may restrict u to the high value as it brings no new information (Figure 5.2 c).

In the final case, we know that j does not hold, therefore its subset cannot hold either, so we can restrict to low(u) (Figure 5.2 d).

The function restricts all occurrences of variables that can be restricted under the assignment. This is the reason why we continue the recursion (returning results of the `FDDRestrict(`high(u) or low(u), j, v`)`) in the 4–18 block. Successors of the node can contain a variable that can be further restricted.

Let us discuss when the search can be stopped. The commands on lines 19–20 serve to optimise the run of the procedure using the properties of variable order. The procedure works correctly even if no order is defined—we defined the < relation to never hold in that case. If class order is used, we may stop the recursion when we leave the relevant class. When total order is defined, this condition acts completely as the procedure for OBDDs described in the literature [Andersen, 1998a]. In all cases, the order of variables is preserved as this procedure may only delete nodes and it never creates new variables.

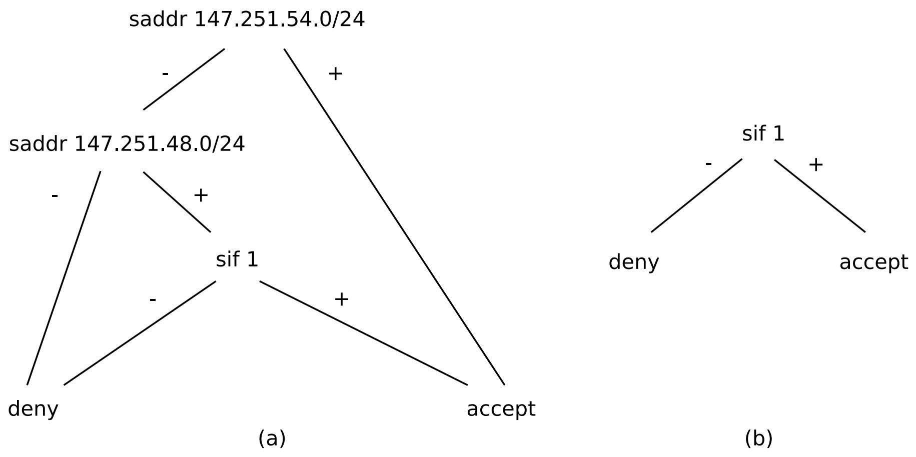


Figure 5.3 Restriction example (by variable `saddr 147.251.48.1/32` to high value)

An example of semantical restriction is shown in Figure 5.3. The diagram (a) is the input. It is restricted by requiring the `saddr 147.251.48.1/32` variable to be high. First, the top node is restricted to its low value, then the `saddr 147.251.48.0/24` is surely satisfied, so it is restricted to its high value. No other nodes can be restricted, so the diagram (b) is the result.

5.3.4 Converting a Filtering Rule to an FDD

Converting a filtering term from F_j into an FDD is straightforward. We rewrite each term t from the rule into an FDD variable with the same test. The high branch of the test leads to the terminal Action(F_j), the low branch to *HSL*. Compare with Figure 5.4. In its upper part, components of filtering rule "`dport 0-1023 saddr 147.251.54.0/24 sif 1 accept`" are shown.

The terms are combined together by FDD function `FDDAnd(`u_1, u_2`)`. See Algorithm 5.3. It computes an FDD equivalent to the conjunction of the terms. This function is based on the standard "Apply" BDD procedure [Bryant, 1992] for computing a logical function of a pair of BDDs. Refinements to this function are related to the fact we use it specially: only to combine FDDs where all terminals are

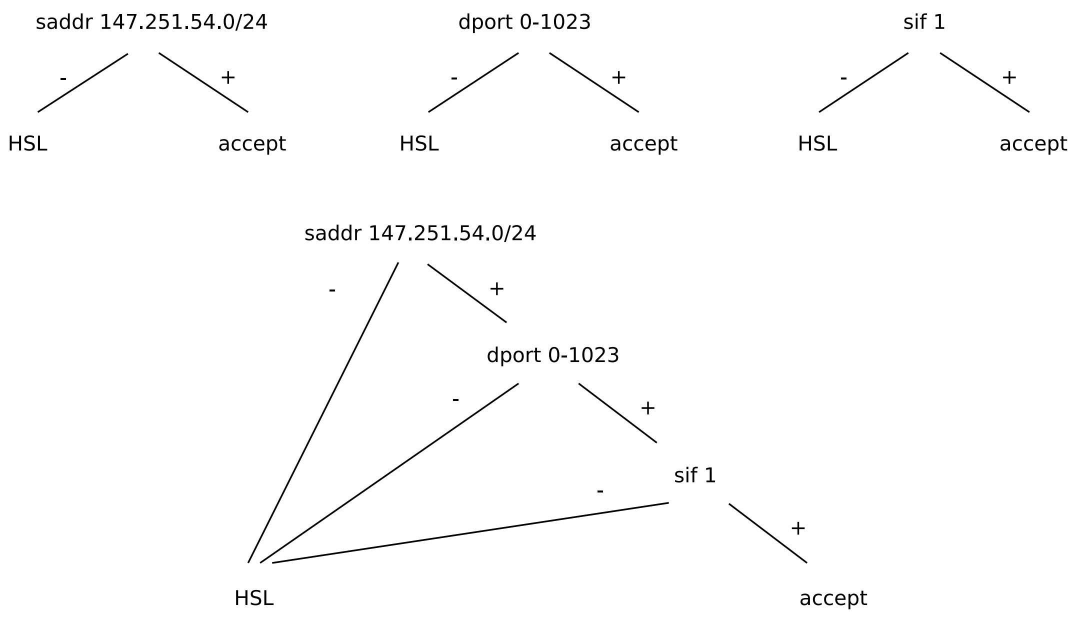


Figure 5.4 Converting a filtering rule to FDD

```
function FDDAnd($u_1, u_2$)
    /* solve terminal cases */
    if $u_1 = HSL$ or $u_2 = HSL$ then return $HSL$
    fi
    if FDDIsTerminal($u_1$) then return $u_2$
    fi
    if FDDIsTerminal($u_2$) then return $u_1$
    fi
    /* perform Shannon expansion on the smallest variable */
    $h$ = smallest of nodes $u_1, u_2$ in relation $<$
    $u_1^h$ = FDDRestrict($u_1$, var($h$), high)
    $u_1^l$ = FDDRestrict($u_1$, var($h$), low)
    $u_2^h$ = FDDRestrict($u_2$, var($h$), high)
    $u_2^l$ = FDDRestrict($u_2$, var($h$), low)
    return FDDCreate($h$, FDDAnd($u_1^h, u_2^h$), FDDAnd($u_1^l, u_2^l$))
end function
```

Algorithm 5.3 FDDAnd(u_1, u_2)

equal or *HSL*—the terminals are taken from a single filtering rule. The advantage of this approach over using the standard and more general Apply function is that we do not have to solve relationship of all terminal symbols.

Supposing that non-*HSL* terminals in u_1 and u_2 are identical, this function is commutative.

```
1  function FDDAppend($u_1, u_2$)
2     if $u_1 = HSL$ then return $u_2$
3     fi
4     if FDDIsTerminal($u_1$) then return $u_1$
5     fi
6     /* perform Shannon expansion on the smallest variable */
7     $h$ = smallest of nodes $u_1, u_2$ in relation $<$
8     $u_1^h$ = FDDRestrict($u_1$, var($h$), high)
9     $u_1^l$ = FDDRestrict($u_1$, var($h$), low)
10    $u_2^h$ = FDDRestrict($u_2$, var($h$), high)
11    $u_2^l$ = FDDRestrict($u_2$, var($h$), low)
12    return FDDCreate($h$, FDDAppend($u_1^h$, $u_2^h$), FDDAppend($u_1^l$, $u_2^l$))
13 end function
```

Algorithm 5.4 `FDDAppend(`u_1, u_2`)`

First, terminal cases are tested. If one of the terminals is *HSL* (meaning "we do not know yet in the filter"), the result is *HSL*, too. If a terminal is reached, we may just append the remainder of the other structure. If a variable order is prescribed and the parameters are ordered, the order is preserved by this step.

We shall precise meaning of line 10 of the algorithm. If no variable order is used, let us understand the choice as "choose the first available variable (say, from u_1)." For class variable order, we choose a variable from the lowest available class. For total variable order, the possibility of choice is abandoned completely—we have to take the lowest available variable. Moreover, as all new nodes are created with the `FDDCreate(`n, l, h`)` function, the resulting structure is reduced. Regardless of variable choice, Shannon expansion is used to propagate the computation to child nodes.

In higher-level procedures, we use notation `FDDConvertRule(`f`)` for the function that converts a firewall rule f into an FDD by means of applying methods described above. It returns the root of the FDD representing rule f.

5.3.5 Converting a Rule Set to an FDD

We have described a function to convert a single rule to an FDD. We have introduced the *HSL* terminal to mark a spot in the FDD where the filter has not decided so far. Suppose we have an FDD representation of filtering rules from the first up to rule i. We will now present a procedure to add rule $i + 1$ to the FDD.

Function `FDDAppend(`u_1, u_2`)` shown in Algorithm 5.4 searches for the *HSL* terminal in the u_1 FDD and replaces it with u_2. It is also the principle of handling

```
$u = HSL$
for $i = 1$ to Size($F$) do
    $f$ = FDDConvertRule($F_i$)
    $u$ = FDDAppend($u$, $f$)
done
```

Algorithm 5.5 Converting a filter to the FDD representation

terminal cases in the algorithm. If *HSL* is found in u_1, it is rewritten to u_2. When we reach another terminal, we return it.

The propagation through the structure is again done with Shannon expansion. Discussion of relationship of variable order to the expansion is completely the same as for function `FDDAnd(`u_1, u_2`)` described in Section 5.3.4.

Converting the whole filter to an equivalent FDD representation is shown in Algorithm 5.5. We start with the *HSL* terminal and we apply rules one by one. We shall show that the resulting FDD is finished (i.e., it does not contain the *HSL* terminal). This is ensured by the fact that the last filtering rule is the default rule containing just the action. Adding the last filtering rule, the *HSL* terminal of so-far-processed filters is rewritten into the default action (it may have been rewritten earlier if some of the preceding rules was default—in that case the computation could have been stopped at that point as all subsequent rules are unreachable anyway).

5.3.6 Implementation Notes

In a practical implementation, the length of addresses is too large to treat the addresses as single entities in hardware, mostly in case of IPv6. Therefore the addresses are split into sequences of registers and the registers act as FDD classes in FDD processing. It has no effect on the functions themselves so we decided to hide this detail in the theory in order not to add extra complexity to the text.

5.4 Combining RA and Filtering into Single Operation

We have described combining routing and ARP tables in Chapter 4, resulting into the RA table represented in a trie. In Section 5.3, we discussed converting packet filters into FDDs. In this section, we will combine both the structures together in order to prepare a single lookup structure that can be converted to the hardware engine.

The basic idea is to *distribute the packet filter into the RA table*—to distribute relevant filters to relevant places of the table according to destination address space specification in filtering rules. The method is based on the following observation.

Theorem 5.6 Principle of filter distribution into address space
Let $p \in IP$ and F be a packet filter. Let $j_1, \dots, j_k$ be a subsequence of all indices from $1, \dots, \text{Size}(F)$ such that $p \in [F_{j_i}]$ for $1 \leq i \leq k$. Let F' consist of rules F_{j_i} for $1 \leq i \leq k$. Then filters F and F' give the same results on all packets with destination address p.

Proof
Packets with destination address p would not match rules specifying another destination address space. □

We maintain an FDD representation of the filter *for each prefix* in the RA trie. Although it may seem as an extreme waste of memory, we argue that all the FDDs are stored in a single hash table and common parts of them are therefore shared. From implementation point of view, the RA table terminals contain pointers to appropriate FDD roots. Formally, we add an extra output to the RA table (compare with Definition 4.1).

Definition 5.7 RA table FDD output
For routing-ARP table RA, let $\text{FDD}(RA_i)$ be the FDD connected to the RA_i record for $1 \leq i \leq \text{Size}(RA)$.

As opposed to the Section 5.3 that was explained bottom-up, this topic should better be presented top-down. Let us have a routing-ARP table RA and a packet filter F. Let us suppose that $\text{FDD}(RA_i)$ is initialised to HSL for all prefixes in the RA table. The main computation loop of the process of applying the filter on the RA table is shown in Algorithm 5.6.

Lines 2–9 serve to determine the destination address space of the filtering rule. If the filtering rule contains destination address specification, we remove it from the rule and we remember the prefix. Otherwise, the rule applies to the whole address space. The destination address space taken from the rule will be expressed by the RA trie prefix.

Function `PrepareSubtrie(`p`)` searches for the prefix p in the RA trie. Following two situations are possible.

1. The prefix p is present in the RA trie. No changes are necessary.

2. The prefix p is not in the trie. Let p' be the longest prefix of p that is present in the trie. In this case we create a new entry for p in the trie and we copy the outcome from p' into the p entry.

```
for $i = 1$ to Size($F$) do
    if rule $F_i$ contains daddr
        then
            let $p$ be the address prefix from $F_i$ daddr
            let $f$ be $F_i$ with daddr removed
        else
            let $p$ be the default route (i.e., the RA trie root)
            $f = F_i$
    fi
    $f' =$ FDDConvertRule($f$)
    PrepareSubtrie($p$)
    ApplyRuleToSubtrie($f'$, $p$)
done
```

Algorithm 5.6 Main loop of RAF computation

Such p' always exists as we require the RA table to contain at least the default RA record. Meaning of the RA table will not be changed by this modification. The prefixes that are routed by p would be routed by p' in the original table.

Finally, function `ApplyRuleToSubtrie(`f', p`)` traverses the whole subtrie denoted by prefix p (and including the prefix) and appends the rule f—calling the `FDDAppend(`u, f'`)` for the filter u that was originally in the trie—to all filters associated with prefixes of the subtrie.

By the same argument we used for a single filter conversion, i.e., that the last filter is the default filter, we can see that all filters produced in the structure are finished as the default filter is applied to the whole RA structure.

From implementation point of view, it is likely that a large number of filters in subtries is identical. We can use a cache of results of applying rule f to filters from the subtrie, preventing the need to compute them again.

5.5 Rewriting the RAF Structure to LUP

In the beginning of this chapter (Section 5.2), we have presented a simple and obvious method to combine RA tables and filters into a single operation lookup. The main drawback of this was that we were not able to rewrite the result reasonably into the target architecture. We will describe a procedure to rewrite the obtained RA/FDD representation into a search that is partly performed by a first-match structure—CAM, and by comparison instructions. The procedure dumps RA trie records to CAM and expands each row created by the filter related to the record (Sections 5.5.1 and 5.5.2). In Section 5.5.3, we precise what types of queries are suitable to perform in CAMs. In Section 5.5.4, we demonstrate that the conversion is correct. Finally, Section 5.5.5 describes how comparison instructions that finish searches are computed.

Besides the basic properties of the design we have summed up in Section 5.2.2, we suppose the following. (Again, motivation was described in Chapter 1.)

- The initial RA trie (i.e., address width) fits into CAM.[30]
- A list *CAMList* prescribes *classes of variables* tested by remaining columns of the CAM. We refer to fields of the list as $CAMList_i$ for $1 \leq i \leq$ Size(*CAMList*). An example of the *CAMList* may be `saddr`, `sif`, `proto`. If another header field is necessary to classify a packet, it must be matched by the instructions.
- The (SRAM) lookup instructions may access any field of the headers. The order of testing header fields by the instructions is arbitrary.

Definition 5.8 CAM rows
We describe a *CAM row* by the *CAMRow* symbol. Technically, *CAMRow* is a ternary string of values 0, 1, and don't-care. It contains destination address prefixes and values of header fields prescribed by *CAMList*.

Resulting values are attached to the *CAMRow*:

- $\mathrm{NH}_{\mathrm{MAC}}(CAMRow)$ is the MAC address of the next hop,
- $\mathrm{NH}_{\mathrm{Int}}(CAMRow)$ is the interface to reach the next hop, and
- FDD(*CAMRow*) is the FDD representation of the filter.

We have connected next hop interfaces and MAC addresses directly to CAM rows. We are allowed to do so: we supposed that width of CAM is not smaller than address width. Moreover, if we propagated the values to terminals of attached FDDs, we would need a terminal for each action and (MAC address, interface) pair, increasing the number of FDD terminals rapidly.

We will now describe the process of rewriting the RAF structure into LUP.

[30] Although resolving the destination address by instructions is possible, it would make the algorithms unnecessarily complex. This requirement is reasonable as commercially available CAMs are capable to hold the addresses.

```
Initialise $CAMRow$ to don't-care values
/* traverse the RA trie in non-increasing prefix lengths */
for $i = 1$ to Size($RA$) do
    insert [$RA_i$] into $CAMRow$
    if $RA_i = SW$
        then
            $CAMRow = SW$
        else
            $\mathrm{NH}_{\mathrm{MAC}}(CAMRow) = \mathrm{NH}_{\mathrm{MAC}}(RA_i)$
            $\mathrm{NH}_{\mathrm{Int}}(CAMRow) = \mathrm{NH}_{\mathrm{Int}}(RA_i)$
    fi
    FDD($CAMRow$) = FDD($RA_i$)
    LUPInsertCAMFilter(0, $CAMRow$)
done
```

Algorithm 5.7 Converting RAF structure into LUP

5.5.1 Rewriting the RA Part

On the highest level, we dump the RA records in non-increasing prefix lengths into CAM. See Algorithm 5.7. We start with a CAM row initialised to don't-care values. We insert the prefix from the RA table to the CAM row and copy appropriate output values. Finally, we set the FDD of the CAM row. Intuitively, we maintain the FDD associated with the CAM row as the FDD representing the rest of the filter that shall be performed to resolve packets matching the row. Function `LUPInsertCAMFilter(`0, *CAMRow*`)` adds FDD variables that can be matched in CAM and finishes the CAM row. Its first parameter denotes that no *CAMList* element has been processed so far, the other is the current *CAMRow* containing the destination address prefix.

5.5.2 Converting FDD to First-Match CAM

Function `LUPInsertCAMFilter(`*i*, *CAMRow*`)` (see Algorithm 5.8) computes the content of CAM and its connection to the FDD that finishes resolution of matched headers. It expands variables belonging to classes prescribed by *CAMList* into a first-match structure and connects the lines of the structure to appropriately reduced FDDs that finish the classification. Parameter *i* is the index of so-far-processed *CAMList* entries. Parameter *CAMRow* is the content of currently processed CAM row.

```
1   function LUPInsertCAMFilter($i$, $CAMRow$)
2       $i = i + 1$
3       if $i \leq$ Size($CAMList$) then
4           /* next $CAMList$ field shall be prepared */
5           choose an FDD variable $j$ belonging to $CAMList_i$
6           if no such $j$ exists in FDD($CAMRow$) then
7               LUPInsertCAMFilter($i$, $CAMRow$)
8           else
9               /* prepare CAM rows for both possible $j$ values */
10              $CAMRow^h = CAMRow$ /* incl. output values */
11              $CAMRow^l = CAMRow$ /* incl. output values */
12              FDD($CAMRow^h$) = FDDRestrict(FDD($CAMRow$), $j$, high)
13              FDD($CAMRow^l$) = FDDRestrict(FDD($CAMRow$), $j$, low)
14              insert value of $j$ into $CAMRow^h$
15              /* keep don't-care values in $CAMRow^l$ */
16              /* recursion: insert subsequent fields */
17              LUPInsertCAMFilter($i$, $CAMRow^h$)
18              LUPInsertCAMFilter($i - 1$, $CAMRow^l$)
19          fi
20      else /* terminal cases */
21          write CAM row $CAMRow$ to the output
22      fi
23  end function
```

Algorithm 5.8 LUPInsertCAMFilter(i, $CAMRow$): placing FDD into CAM

In Algorithm 5.8, we first check if the CAM row has been finished. If it is not the case (lines 4–19), we continue the recursion splitting the search on a suitable variable if possible. We will discuss this part of the algorithm in detail.

- First, we try to find a variable that can be resolved in currently processed CAM column (line 5). It must belong to the $CAMList_i$ class. We should choose a variable that really appears in the FDD($CAMRow$) in order to allow the CAM row to be split into two (hence not wasting the CAM space). We may, e.g., traverse the FDD until we find a variable belonging to $CAMList_i$ class.

- If no such variable is found, we cannot use this CAM column to improve resolution of the FDD and the space must be left unused. In that case, we keep don't-care values in appropriate positions of the CAM row and continue to the next *CAMList* field.

- In a case that a suitable variable j belonging to $CAMList_i$ has been found, we shall create two copies of the current CAM row, to find a suitable FDD variable belonging to the $CAMList_i$ class, to compute FDDs for successful test of the variable and the opposite and to continue the recursion in both cases, see lines 9–18.

  – In lines 9–15, we create two copies[31] of the current *CAMRow* including the output values (i.e., $\mathrm{NH}_{\mathrm{MAC}}(CAMRow)$ and $\mathrm{NH}_{\mathrm{Int}}(CAMRow)$ or *SW*). The first copy $CAMRow^h$ will be used for the case that variable j is satisfied and $CAMRow^l$ for the opposite.

    We compute both restrictions of the FDD for j. Note that the resulting FDDs differ (if they did not, the node containing j would not be reduced).

    We insert the test of variable j into appropriate position of $CAMRow^h$. Don't-care symbols stay unchanged in $CAMRow^l$.

    Finally, we continue the recursion for the rows split depending on value of variable j. In lines 17–18, first the CAM row for high value of variable j is created and then the opposite.

    In the case of $CAMRow^l$, the block originating from this expansion is evaluated when variable j does not hold. We can use the same CAM column to insert another possible variable from the same class to test, therefore we call the recursion for $i - 1$. Note that when no suitable variable can be found, don't-care symbols will be preserved here.

The remaining case is that the CAM row has been finished. Then, we just write the *CAMRow* to the output. The algorithm emits CAM rows one-by-one in first-match order.

5.5.3 Query Types in CAM

We presented Algorithm 5.8 hiding the problem that various types of queries differ in price paid in CAM. CAM is suitable to perform exact match and prefix match queries, a test of this type can be performed by a single CAM row, therefore the algorithm works exactly as we presented it.

This is not completely the case of range queries. To perform such queries in general, the range has to be converted into prefixes, i.e., expanded into several

[31] We would not create two copies in a real implementation—we only change the associated FDD, so creating a single copy is sufficient. Optimising the procedure would make it much less readable.

CAM rows [Spitznagel et al., 2003], [Taylor and Spitznagel, 2005]. We can understand creating CAM rows in Algorithm 5.8 as creating *blocks* of CAM rows that cover the range with prefixes and all the rows in the block are handled as a single unit in the algorithm.

If the choice of variable classes is rich enough, such approach would lead to wasting CAM space. Therefore, we should prefer using CAM for other types of queries than range ones. To make the implementation easier, we can solve this problem by a "design decision," we just never include range queries to the *CAMList* and test them by SRAM instructions (we always need up to two comparison instructions to check a range).

5.5.4 Correctness of the CAM Search

We have to show that the output interface and next hop MAC address is the same in the RA trie and in the CAM computed by Algorithm 5.7. It would be obvious if the Algorithm 5.7 did not use Algorithm 5.8 as a subroutine—then, the RA prefixes themselves are filled into CAM in non-increasing prefix lengths, therefore the first matching CAM entry corresponds to the longest matching prefix in the RA trie (using the same principle as in Section 3.1.3).

The question is whether using columns of CAM to represent parts of packet filter does not interfere with the result of routing. In the following theorem, we show that the result of RA lookup (i.e., next hop interface and MAC address) is the same for both RA and CAM generated.

Theorem 5.9 Result of RA is correct in CAM
Let $p \in IP$. Let *CAMRow* be the matching CAM row for a packet destined to address p. Then results of both $RA(p)$ and *CAMRow* are either *SW* or the interface and MAC address of the next hop are identical for $RA(p)$ and the matching *CAMRow*.

Proof
Let q be the longest matching prefix for destination address p in the RA table.

Taking into account the construction of CAM in Algorithm 5.7, we can divide the CAM into three subsequent blocks:

- block A contains prefixes at least as long as q that do not match p (therefore p cannot match in this block),
- block B contains prefix q (possibly repeated several times because of using Algorithm 5.7),
- block C contains the rest of CAM content.

The theorem holds if the packet matches block B as all records in this block originate from the RA prefix q. We will show that packets with destination address p match in block B (regardless of other header fields and packet filter setting).

Algorithm 5.8 either preserves don't-care values or it creates a pair of rows (see lines 17–18). In the latter case, the second row produced keeps don't-care values. Therefore, the very last row produced by the recursion contains only don't-care symbols. The last line of block B contains the prefix q and don't-cares only, so the destination address p must match in block B. □

5.5.5 Creating Lookup Instructions

Each CAM row we produced in the previous section has an associated FDD that finishes classification of packets matched there. The tests in the FDD can be directly rewritten into SRAM instructions. We stored all FDDs in a single structure—a hash table keeping their nodes. We can rewrite the hash table into SRAM instructions. For each CAM row *CAMRow*, we only arrange the computation to jump to appropriate root of FDD(*CAMRow*).

This approach does not insert unreachable parts of the structure into the instructions. To show this, we have to indicate how node storage is implemented. FDD nodes are stored in a hash table and each node has an associated value denoting "how many times" the node is used in all structures represented by the hash table. We only have to delete FDDs that are not used anymore. This is done decreasing the usage counter of appropriate nodes. When the counter reaches zero, the node can be removed. In Algorithm 5.8, this situation occurs when the restricted FDDs for $CAMRow^h$ and $CAMRow^l$ are computed. Then the FDD associated with the original *CAMRow* can be deleted, so parts of the structure that became unreachable are removed.

5.6 Complexity Considerations

Let us suppose that the maximum number of terms in a filtering rule is n. In the operating system, rules are evaluated one-by-one. Taking evaluation of a single term as a unit of complexity, we need $O(\text{Size}(F) \cdot n)$ operations in the worst case to test a packet against the filter.

In the FDD as described in Section 5.3, the estimate of time complexity can be easily seen when we relax variable ordering and we do not even require the FDD to be reduced. In that case, traversing the FDD is completely the same as evaluating the filter in software lazily (i.e., when a term does not hold, we go immediately to the subsequent rule). Therefore the theoretical worst case complexity is still $O(\text{Size}(F) \cdot n)$. Practically, properties of reducedness and variable ordering find identical nodes and unify them, decreasing the number of tests needed.

The number of tests is moreover reduced by the method of distributing filters into target address space (Section 5.4). We (1) do not test destination address space again in the filter and (2) we do not include filters that would not match anyway because of another destination address space specified.

As we have shown in Section 4.2.6, time to evaluate the RA table in software is $O(w)$ where w is bit length of the address. By rewriting the structure into CAM, we reduce it to $O(1)$ by means of brute-force hardware. Presuming that the RA structure fits into CAM, the whole evaluation of routing, ARP, and filtering can be therefore computed in $O(\text{Size}(F) \cdot n)$ time. This time is necessary either to classify a packet or to decide that the packet cannot be classified by the hardware engine and must be sent to the operating system. In any case, this time is not worse than time necessary for software classification.

As the CAM computation and evaluation of comparison instructions can be pipelined, we get the whole time spent in CAM "for free."

Theoretical worst case space complexity estimate is the same as for OBDDs—exponential [Bryant, 1995]—as can be seen from the algorithms we used for constructions, they use recursive calls based on Shannon expansion.

5.7 Packets Delivered to the Host Computer

We have discussed handling forwarded traffic only. The remaining problem is twofold.

1. The host machine can contain network interfaces that do not belong to the card, so traffic forwarded through them cannot be switched by the accelerator and must be delivered to the operating system, too.

2. We shall show how traffic destined to the host computer can be delivered there and how it can be passed through a hardware accelerated input packet filter.

To take into account that not all interfaces belong to the accelerator, we have to change the construction of RA table described in Section 4.2.3. During computation of the RA table, we test if the interface prescribed by the route we process belongs to the accelerator. If so, we use the original algorithm. In the opposite case, we just insert a routing-ARP record destined to software—the packets matching this record must be forwarded to an interface outside the accelerator. Note that all arguments on correctness of the method remain valid as we considered sending a packet to the operating system as a correct result.

To solve delivering traffic destined to the host computer, we have to create a list of all IP addresses configured on interfaces of the accelerator. We insert those addresses into a special RA table and set their destination to software. We expand this RA table into the beginning of CAM as a completely independent part.

We combined a *forwarding* filter with the forwarding RA table (see Section 3.3.3 for the discussion on various types of filters). With the same method, we can apply an *input* filter to the RA table containing own addresses. As we have discussed in Section 3.3.3.3, applying the input filter is not mandatory as the packets are filtered by the operating system anyway. It can nevertheless turn out useful in case of attacks—we can discard unwanted traffic in the accelerator, avoiding system bus overload.

From implementation point of view, both the forwarding filter and the input filter applied on traffic destined to the host computer can be stored in a single hash table. It allows processing both filters in a uniform manner, moreover saving memory as common nodes of the filters are shared.

5.8 Dealing with Implementation Limitations

One of problems we deal with in an implementation is that CAMs are quite limited in their sizes. We therefore have to give methods how to arrange the structures in a reasonable manner to deal with such a limitation.

We have designed the method of rewriting the RAF structure into CAM based on a single requirement: that the width of CAM is capable to contain destination addresses. The rest of the method (i.e., Algorithm 5.8) adapts to width of CAM according to the configuration of *CAMList* parameter.

The limit that can be met easily is the number of records in the CAM. The easiest solution to the situation when we ran out of CAM rows during completing CAM content is based on principles of software cooperation described in Section 4.1. We just stop building CAM content and insert a final row containing don't-cares only destined to *SW*. The main drawback of this method is that it is extremely rough—it does not take into account that majority of traffic could be handled by records we did not insert into the structure, forcing the operating system to handle it, not using the potential of the space in the memory we wasted to keep much less frequently used records.

One of the most beautiful properties of the Algorithm 5.8 is that it is very adaptable. We can save space of CAM regulating expansion of each row separately. We can stop the recursive expansion of the variables into CAM at any level, the result still behaves correctly. If some knowledge of traffic distribution over RA records is available, we can prefer expanding most frequently used entries over less used ones.

Similar situation can occur with instructions in SRAM. Any FDD subgraph can be removed and replaced with a terminal sending all the traffic to the operating system.

The form and size of the resulting lookup structure depends heavily on the *CAMList* parameter and ordering of variables in the FDD. We leave investigating methods of setting both the parameters as a future work. Effects of setting the *CAMList* parameter and other quantitative measures will be presented in the following Chapter 6.

6 Experiments

Personnel operating the equipment must be trained and competent; must not conduct themselves in a careless, willfully negligent, or hostile manner; and must abide by the instructions provided by the documentation.
– [Juniper Networks, Inc., 2005a]

While time complexity estimates of FDD based RAF structures suggest that they shall behave well, their overall size complexity is exponential. On the other hand, BDD-based structures are known to behave "much better" compared to their exponential theoretical limit in practice.

In order to evaluate usability of the FDD based RAF structures, we study them experimentally in terms of

- memory usage,
- time required to perform lookup.

Structures used for evaluation have been obtained using a prototype implementation of the method.

Evaluating the structure in hardware, we would obtain absolute values and timings. On the contrary, the results would be very difficult to obtain and to interpret—they would depend not only on the packet classification unit itself, but also on the rest of the hardware design. Anyway, the complete hardware design that would allow testing the overall performance (from input to output port) and compare it to overall performance of a software router is not available at the time of writing. When software simulation is employed, we get hardware-independent results that can be easily parametrised by properties of the actual accelerator.

Section 6.1 describes the simulation environment. Data for experiments have been obtained from routers serving in a production network, issues related to source experimental data are presented in Section 6.2. The guide to interpretation of the results is given in Section 6.3. Section 6.4 contains selected and commented results. Detailed results are shown in Appendix B. The concluding Section 6.5 summarises the results.

6.1 Simulation Environment

A prototype implementation called `lupgen` (LUP GENerator) has been used to generate RAF structures out of source tables. Properties of the structures have been collected and real packet traces have been simulated by Perl scripts.

The packet filtering language used as `lupgen` source has been presented in Figure 3.6 in Section 3.3.5. It contains all types of queries, i.e., prefix, range, and exact matches.

The scenario for measurements in this chapter is usually the same: we compute the RAF structures out of input data with `lupgen`, determine its "static properties" (e.g., size of memories occupied), and compute behavioural characteristics on a set of packets.

Software simulation is feasible, unfortunately, it has a disadvantage: the measurements are performed on a snapshot of tables only. In the reality, a packet that shall be delivered to a node with currently unknown ARP record is sent to the operating system and cause ARP protocol to be initiated. The change in the ARP table causes the whole structure to be recomputed, so the previously unaccelerated traffic gets processed in the accelerator. This behaviour is not reflected in the simulation, we just take initial value of the ARP table and the table is unchanged thorough the experiment. Hence, it does not evaluate if the packet is accelerated or it has to be sent to the operating system because of a missing ARP record. The simulation is not able to measure the ratio of packets handled by the accelerator and sent to the operating system to be processed.

In the simulation, we assume potentially infinite memories, i.e., we have memory capable to contain the structures however large they are. We do not bound to concrete implementation limitations; none of the techniques of dealing with them (as we have described in Section 5.8) has been incorporated to the simulation environment. It makes experiments independent on an actual implementation and sizes of memories used in such implementation, such results may also serve to determine suitable hardware memory sizes. The numbers obtained are "best" in the sense they prefer using CAM rows for a given CAM width. Limiting the number of CAM rows and employing adaptive techniques would move parts of the computation into the FDD part.

Besides routing and ARP tables and the packet filter setting, results also depend on the following parameters:

- allocation of CAM columns (i.e., parameter *CAMList* described in Section 5.5); allocation of columns also describes CAM width used,
- FDD variable order used internally in the generator.

Data set	# of routes	# of ARP	# terms in filter	# of filters
1	643	623	105	56
2	889	102	80	35
3	889	162	80	35

Table 6.1 Characteristics of Measurement Sources

Allocation of CAM columns will be chosen uniformly for most experiments to make them comparable and one of the experiments studies dependency of the resulting structure on *CAMList*. Order of FDD variables is based on class order depending on *CAMList* in all cases (cf. Section 5.3.1 for variable order discussion).

6.2 Experimental Data

The sources we need for measurements are basically a routing table, ARP table, and a packet filter setting. Moreover, a sample of traffic will be used to evaluate the structures in real conditions. The question is how and where to obtain such data.

Various routing table and traffic samples are available (e.g., [Andersen et al., 2001], [Feamster et al., 2003], [Claffy, 1999]). Alas, to the author's knowledge, no publicly available source exists that contains all data we need for our measurements. Especially security administrators are extremely careful to keep their access lists secret.

Generating measurement sources randomly is not able to produce realistic results. None of the sources is random. Routing tables depend mainly on network topology and partly on traffic, ARP tables depend on routing and recent history of traffic, packet filters reflect security policies (which is again based on expected and real traffic) and are highly structured.

For the reasons above, the author decided to obtain real-world routing and ARP tables, packet filters, and packet traces. The data sets have been extracted from edge routers connecting mid-sized academic networks to the national academic backbone[32]. As the data originate from production networks, the author is not allowed to disclose details of packet filters and packet traces.

Three data sets have been prepared. Their properties are summarised in Table 6.1. For each data set, number of routing records and ARP records is shown in the table. The number of terms is the total number of filtering language terms in each filter (excluding actions), the last column contains the number of filtering

[32] The author would like to thank Petr Adamec and David Rohleder for kindly providing the data.

rules. Packet filters in all sets are typical "permissive" ones, they accept everything except what is explicitly forbidden.

Data sets 2 and 3 originate from boxes that share load of the same network. Their routing tables are very similar and their packet filters are completely identical. We use them to examine whether the structures really behave in a similar manner for similar inputs. Data set 1 originates from a different autonomous system than sets 2 and 3.

In the experiments, we need several lengths of packet filters and routing tables. A packet filter of length n have been obtained taking $n-1$ rules from the beginning of the filter and adding the very last rule (which is the default rule containing just the action `accept` for permissive filters) in order to make the filter total. The reason of taking the "beginning" of the filter is twofold. First, the order of rules is significant, we cannot choose rules in random. Second, filtering rules are handmade, we can assume that some care have been taken in choosing rule order [Attar, 2002]. In case of routing tables, no special route order is required, we therefore take the specified number of rules in random, only taking care that all sets produced contain the default route.

6.3 General Notes on Experimental Results

In all experiments, we obtain two types of results:

- "static" characteristics of the generated structures, i.e., sizes of memories and minimum/maximum lengths of FDD traversals,

- "dynamic" characteristics, i.e., behaviour of the RAF structure representation on a traffic sample.

Minimum and maximum lengths of FDD traversals describe time required to evaluate a packet against the structure. As we can see from the algorithms constructing the structure, the smallest minimum value of FDD is 1, in other words, at least the terminal FDD node must be visited.

In practice, units performing CAM lookups and interpretation of instructions can be pipelined, therefore a constant covering the CAM lookup time can be subtracted from the FDD traversal result (we obtain some time "for free" thanks to pipelining). However, we will not include pipelining into the results as the actual number of FDD evaluations that can be performed during CAM lookup is hardware implementation dependent.

Dynamic characteristics of the structure have been obtained simulating lookups of packets of the traffic sample in the structures and counting depths where

Value	Meaning
`Proto`	Protocol
`SrcIP0`	Source IP address 16 most significant bits
`SrcIP1`	Source IP address 16 least significant bits
`DstIP0`	Destination IP address 16 most significant bits
`DstIP1`	Destination IP address 16 least significant bits

Table 6.2 Values of *CAMList*

terminal actions were reached. The result is a histogram of number of packets requiring certain number of FDD steps to be classified.

Characteristics of the structures depend on setting of *CAMList*, i.e., prescription what parts of packet information shall be handled by CAM. The total width of fields prescribed by *CAMList* must not exceed the width of the CAM.

The testing packet filter supports protocols, source and destination addresses, and source and destination ports. As handling port numbers is quite complex in CAM as we have discussed in Section 5.5.3, the decision was made not to allow port numbers there. All port tests must be performed by the FDD. In the implementation, the FDD variables are not exactly the same as in the theoretical description, addresses are split into two pieces indexed with 0 and 1 containing 16 most significant and least significant bits of the address. Possible members of *CAMList* are shown in Table 6.2.

6.4 Experimental Results

In the experiments, we keep all sources except one fixed throughout a series of tests. The experiments are divided into three parts: variable filter length (Section 6.4.1), variable routing table length (Section 6.4.2), and effect of changing the *CAMList* parameter (Section 6.4.3).

All groups of experiments consist of studying static properties of structures generated and tests performed on traffic sample of 75,000 packets captured on the same box the source tables were obtained.

The setting of all experiments is *CAMList* = `Proto`, `DstIP0`, `DstIP1`, `SrcIP0` unless said otherwise. Characteristics of the data sets and the way how lists of variable lengths were obtained have been given in Section 6.2.

6.4.1 Variable Filter Length

In the first set of experiment, we study how packet filter size affects the resulting structures.

6.4.1.1 CAM Size

In Figure 6.1, we can see how number of CAM rows depends on packet filter length for all data sets. Big increases in the number of CAM records correspond to blocks of source address tests in the filters that caused expansion of large blocks of CAM rows by testing `SrcIP0`. For example, in sets 2 and 3, the only rules of the form `deny saddr` for large networks that do not contain additional conditions (and they therefore apply to the whole structure) appear between rules 19 and 24.

Sets 2 and 3 have identical filters and very similar routing tables. They were expected to produce similar results. Indeed, their results differ in order of ones, so in the resolution of the graph, the difference is invisible. Higher number of CAM rows for the set 1 is caused by the number of source address tests related to the complete destination address space that are expanded into the CAM.

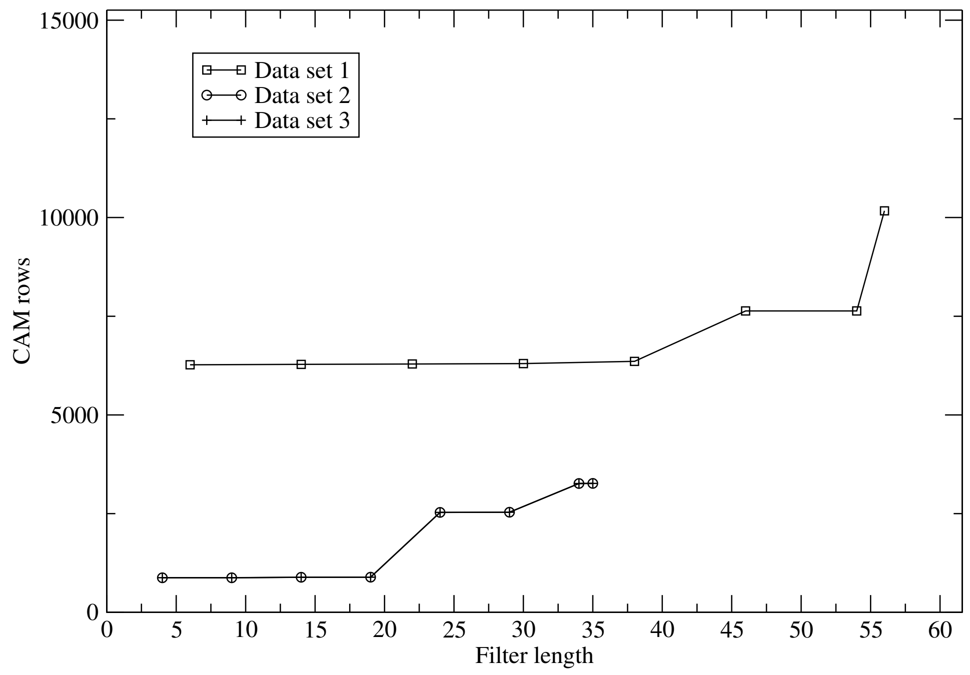


Figure 6.1 Number of CAM rows depending on filter length

6.4.1.2 FDD Nodes

Figure 6.2 shows the number of FDD nodes (i.e., the size of memory) needed to represent the filter for varying filter lengths. The shallow characteristics of sets

2 and 3 corresponds to the fact that those filters tend to test port ranges combined with other (mostly destination address) conditions and the number of distinct port range tests is small. On the opposite, set 1 contains a number of various port range tests—partly bound to destination addresses—starting from rule 30.

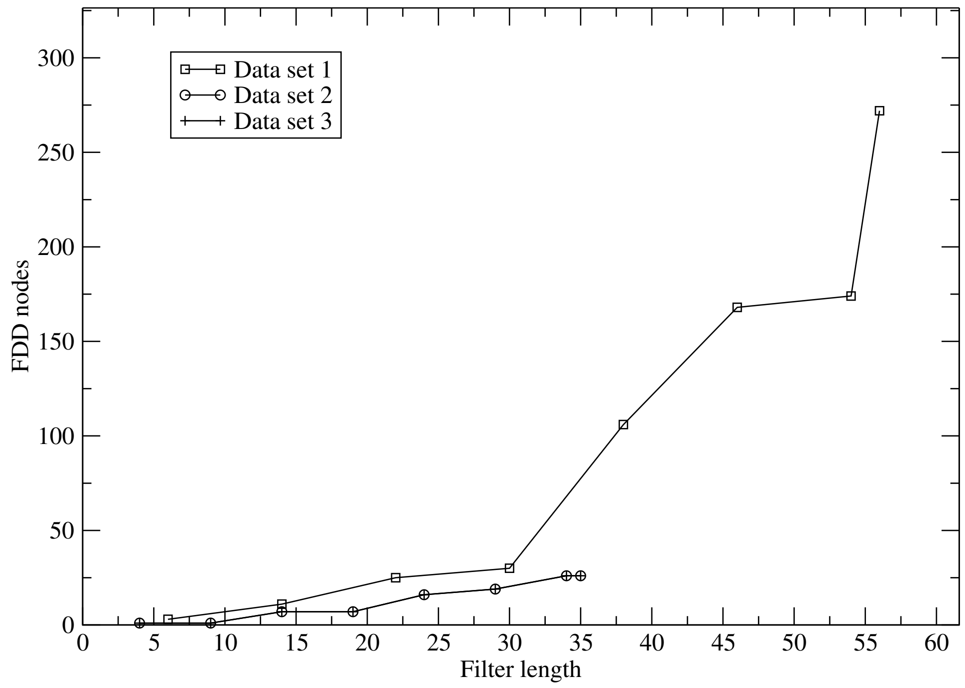


Figure 6.2 Number of FDD nodes depending on filter length

6.4.1.3 FDD Paths

The minimum number of FDD tests when evaluating a packet is 1 (as follows immediately from the construction of memories). Such paths were reachable practically in all structures we obtained, therefore, we do not present shortest paths graphically. It occurs in cases when CAM contains all classes necessary to classify a packet.

If the content of CAM is not sufficient to perform the classification, the important value is length of the maximum path traversable in the FDD structure, which is also the maximum time necessary for the classification. This value is theoretically bound from above by the total number of filtering terms in the filter.

In Figure 6.3, we can see that practical values are strongly below the theoretical maximum, e.g., maximum depth for data set 1 is 21 for a filter with 56 rules containing 105 filtering terms.

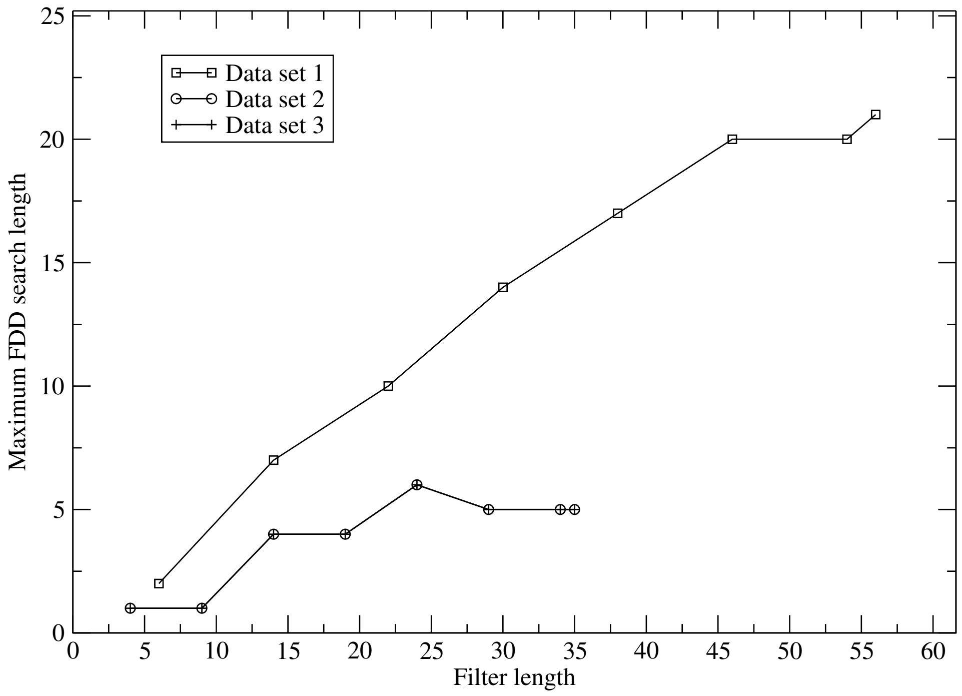


Figure 6.3 Maximum paths in FDD depending on filter length

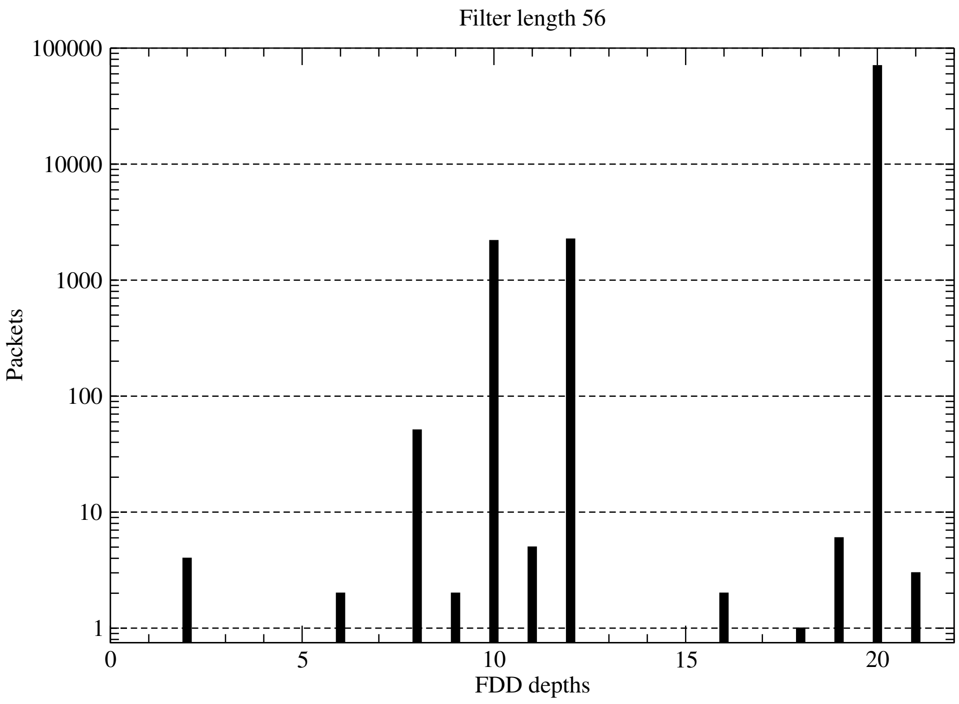


Figure 6.4 Histogram of FDD depths for data set 1 (56 filtering rules)

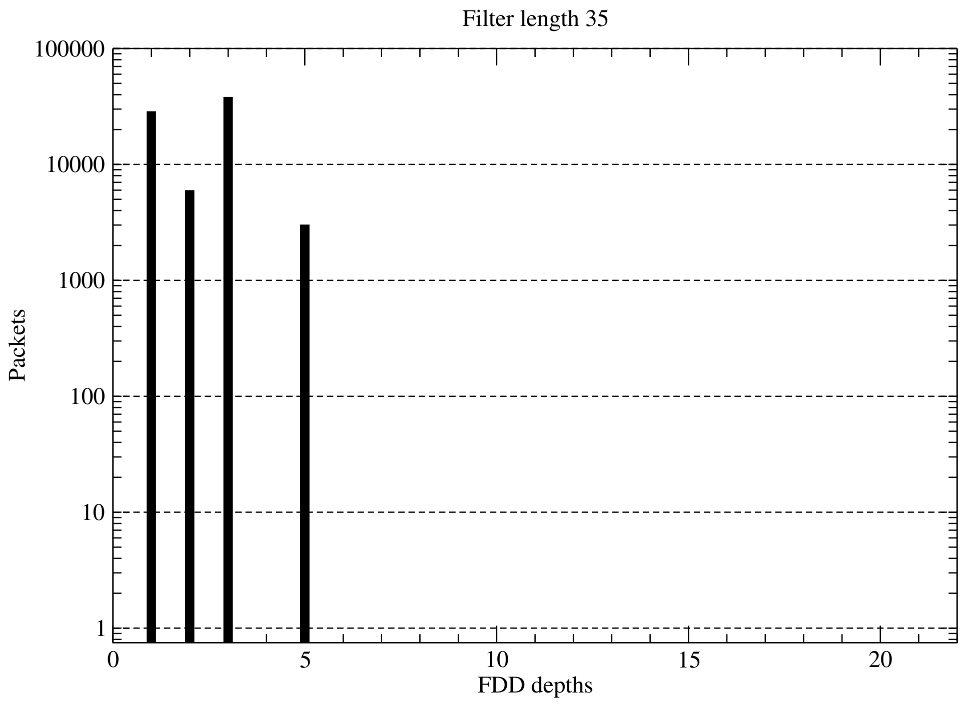


Figure 6.5 Histogram of FDD depths for data set 2 (35 filtering rules)

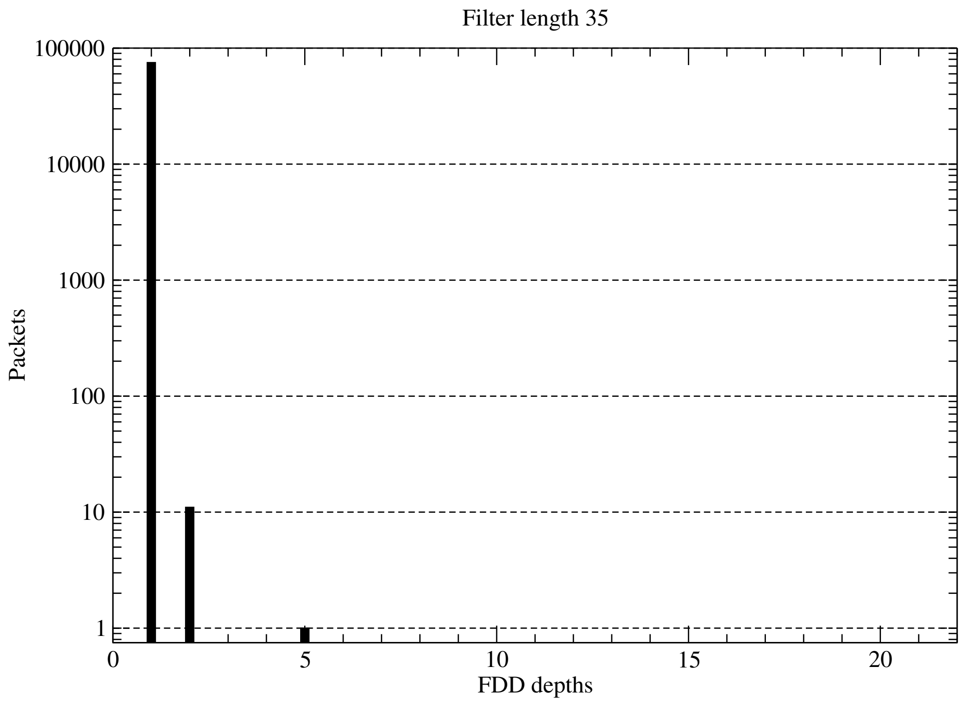


Figure 6.6 Histogram of FDD depths for data set 3 (35 filtering rules)

6.4.1.4 Traffic Samples

For brevity, only several examples of traffic processing (the ones with full filters) are presented in this section. Complete results can be found in Appendix B, Section B.1.

The histograms describe how many packets out of the sample of 75,000 is resolved in a certain depth of the FDD. In the first data set (Figure 6.4), majority of the traffic needs 20 FDD tests. We can see small peaks at levels 10 and 12. Because of differences in orders of magnitude, we present the histograms in logarithmic scales.

The graph in Figure 6.5 preferred significantly shorter lengths. The result in Figure 6.6 has maximum FDD path of 5 steps and vast majority of packets in the sample was resolved in just one step (i.e., by CAM only).

In general, results obtained correspond to the principle of network traffic locality: in packet samples, several large data streams (that result into identical actions in the filter) can be discovered.

6.4.2 Variable Routing Table Length

This section studies effects of routing table length on characteristics of resulting structures, presuming all other sources are fixed in a particular testing data set (especially, we take complete packet filters).

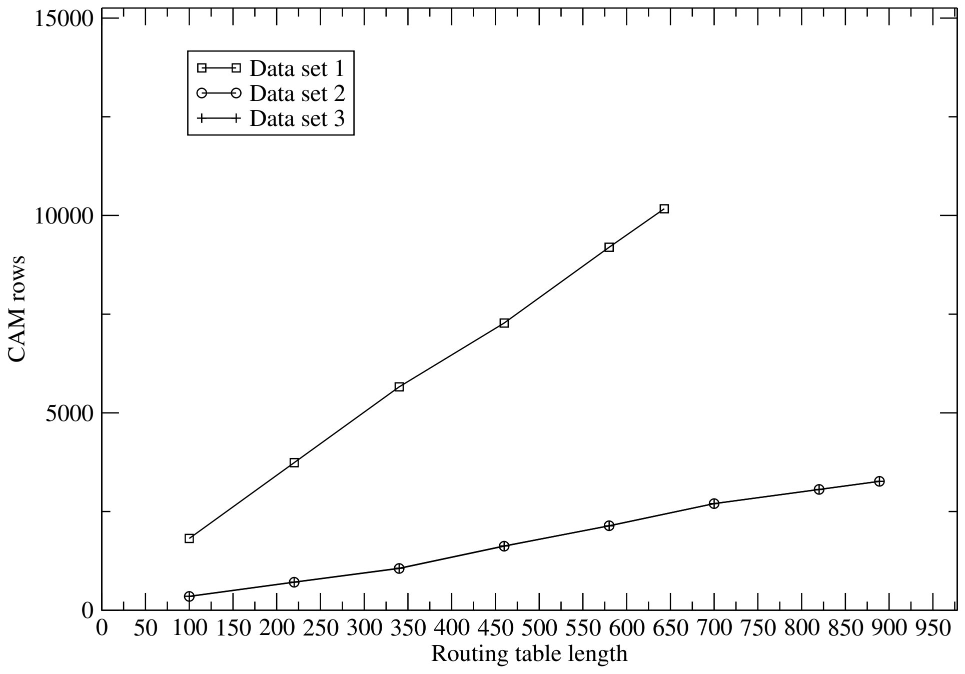


Figure 6.7 Number of CAM rows depending on routing table length

Name	*CAMList* value
A	`Proto`, `DstIP0`, `DstIP1`
B	`Proto`, `DstIP0`, `DstIP1`, `SrcIP0`
C	`Proto`, `DstIP0`, `DstIP1`, `SrcIP1`
D	`Proto`, `DstIP0`, `DstIP1`, `SrcIP0`, `SrcIP1`
E	`Proto`, `DstIP0`, `DstIP1`, `SrcIP1`, `SrcIP0`

Table 6.3 Effects of *CAMList*—*CAMList* shorthands

As we can see in Figure 6.7, the size of CAM is approximately linear with respect to the length of routing table. Slight variations are caused by expansion ARP records in local networks and by expanding FDD variables into CAM.

We have experimentally verified that size of the resulting FDD as well as maximum FDD path length do not depend on routing table size at all, supposing that the routing-ARP of the structure fits into the CAM.

As we used the full packet filters for all data sets, the results presented in the previous Section 6.4.1 for maximum filter length hold independently on routing table size. The same is true for traffic measurements, the reader can refer to Section 6.4.1.4, the histograms depicted there contain full length filters, they are also valid for all measured routing table sizes.

6.4.3 Effect of CAM Allocation

The resulting structure depends strongly on the form of the *CAMList* parameter, i.e., prescription what parts of packet information shall be processed by CAM. In practice, the limiting factor for *CAMList* choice is the width of the CAM and partly also the fact that range queries are difficult to express in CAM (cf. Section 5.5.3). We have left the question how *CAMList* shall be chosen as future work. To demonstrate that a suitable method of *CAMList* choice is crucial, we give measurement results for several possible *CAMList* values.

Table 6.3 gives an overview (and symbolic names defined there we will use for brevity) of the choices of *CAMList*s we used for the experiment. Note that *CAMList* must contain at least destination addresses. In our implementation, a register describing level 3 flags must be also included in CAM (in order to handle erroneous packets), therefore protocols must be also included; *CAMList* A is the absolute minimum. Lists B and C add one half of source address, the remaining lists insert whole source addresses into CAM. Lists D and E have the same widths, only high and low parts of source IP addresses are swapped.

CAMList	CAM rows	FDD nodes	FDD min path	FDD max path
Data set 1				
A	1,272	143	2	29
B	10,170	272	1	21
C	10,176	382	1	26
D	19,054	129	1	20
E	63,496	129	1	20
Data set 2				
A	1,128	49	1	11
B	3,264	26	1	5
C	5,050	27	1	6
D	7,151	7	1	3
E	14,934	7	1	3
Data set 3				
A	1,128	49	1	11
B	3,264	26	1	5
C	5,050	26	1	6
D	7,151	7	1	3
E	14,934	7	1	3

Table 6.4 Effects of *CAMList*

Results are presented in Table 6.4. For each data set and for each *CAMList* choice, the table contains number of CAM rows needed, number of FDD nodes, and minimum and maximum path traversable in the FDD.

We can see that adding fields to CAM, length of the maximum traversable FDD path tends to decrease. Comparing *CAMList*s B and C and mainly D and E, we observe that preferring higher parts of source addresses in CAM leads to significantly better results in terms of CAM rows needed and slightly in lengths of FDD paths. This is clearly visible mainly for variants D and E in the data set 1 when length of CAM needed increases more than three times only because of variable order choice. Methods of suitable variable order choice need further investigation.

Traffic sample results for variable *CAMList* are presented in Appendix B.2.

6.5 Evaluation of Experimental Results

The results presented in this chapter are not bound to a real implementation. To fulfill the plan we stated in the beginning, i.e., to evaluate FDD based RAF structures in terms of memory usage and lookup time, we shall discuss that the structures are usable in the real world. We split the discussion into two parts, memory usage in Section 6.5.1 and time properties (Section 6.5.2).

6.5.1 Memory Usage

Size of available CAMs is in order of megabits. A 2 Mbit CAM can be configured to the width of 136 bits (which is sufficient to contain all *CAMList* configurations we used), having 16,384 rows. In this configuration, most of the experiments fit into such CAM. Cases when the required number of CAM rows exceeds reasonably the size of memory can be solved by methods described in Section 5.8, using "inner scalability" of the method.

CAMs up to 18 Mbits are available on the market, which significantly exceeds memory requirements of the method with realistic testing data sets. Those devices are configurable to various widths. Further research is needed to find a suitable way to set the trade-off between CAM width and number of rows. The method does not depend specifically on size of a single CAM, more CAM units can be employed in the following way ("outer scalability" of the method). Let us consider configuring a pair of CAMs "one below another." If the first unit matches, its result is used. In the opposite case, the result of the second unit is taken. Both searches can be performed in parallel, thus time requirements are the same as for a single chip. This approach can be generalised to more units than just two.

The major problem is not the CAM size but the choice of a suitable variable order, as we can see in the experiment with variable *CAMList* (cf. Table 6.4). A method (at least a good heuristics) to find a suitable order is necessary.

In order to interpret the FDD part of the structure, FDD nodes are rewritten into the form of comparison instructions in a static RAM. We are safely below limit of such memories, at most hundreds of FDD nodes were occupied in the experiments, whereas hundreds of thousands instructions can be held in SRAMs.

To summarise, the results produced in the experiments are usable in contemporary memories provided that suitable variable order is chosen. Moreover, the method allows scaling the design by adding CAM units.

6.5.2 Lookup Time

As the CAM unit and the unit interpreting FDD nodes are pipelined, the only measure we need to evaluate in order to obtain the throughput of the classification unit is the number of instructions executed, in other words, the number of FDD nodes traversed in the lookup.

The time complexity of FDD traversal is limited from above (see Section 5.6) by the total number of terms in the filter. In the worst experimental result we obtained, the filter with 105 terms have been evaluated at most in 28 FDD comparisons, i.e., with a constant multiplicative factor 0.27 to the theoretical measure. It happened in a configuration where the *CAMList* contained just the minimum configuration, wider CAMs improved the factor significantly.

It is not trivial to find a part of the software router to compare to the classification engine in terms of lookup time. The way of packet classification in the operating system is completely different from the hardware, moreover, parsing headers (which is done by a pipelined hardware engine in the accelerator) is distributed among classification operations. We would not obtain a reasonable comparison measuring time necessary to process a packet in the kernel. Measuring the overall performance of the hardware accelerated system would evaluate the system as a whole, not the classification engine. In any case, the complete hardware design is not available at the time of writing this work.

To put the measurement onto a real basis, we will give an estimate of throughput of the hardware classification engine based on properties of the prototype implementation of LUP and compare it to maximum theoretical throughput of a PC router.

The PC architecture equipped with a 64 bit/66 MHz PCI bus has theoretical bus throughput 4 Gbps. The highest theoretical traffic that can be handled by such a router is 4 Gbps, considering the state where all incoming packets are dropped by the filter.

In the prototype implementation of the hardware design, a CAM search is performed in 160 nsec, a SRAM instruction takes 40 nsec.[33]

If we omit results obtained for unreasonably narrow CAMs (i.e., for *CAMList* setting A), we may take 20 as an estimate of maximum number of FDD steps. Classifying a packet then takes 800 nsec, thus 1,250,000 packets are classified per second.

For 1500 B packets (omitting interpacket gaps), it corresponds to 15 Gbps data stream. For shortest packets, i.e., 64 B, we obtain a stream of 640 Mbps. While this seems bad compared to the PC, we must note that the PC would not be able to handle such a traffic anyway, the number of interrupts necessary is unfeasible.

[33] A reimplementation of the lookup engine with expected SRAM instruction execution time about 20 nsec is under development.

Compare with [Oppermann, 2006], highly optimised FreeBSD code is capable of processing up to 750,000 packets per second.

7 Conclusion

The goal of this work is the proposal of the method to combine routing, level 3-to-level 2 address translation, and packet filtering into a single lookup machine balancing computation between CAM and comparison instructions, and between the hardware engine and the operating system.

We have proposed the method formally, using a notion of filtering decision diagrams to represent packet filters. Besides formal evaluation of the method by means of complexity estimates, measurements based on software simulations have been performed. The resulting method has been shown feasible to be employed in the contemporary hardware and efficient to outperform packet classification possibilities of the PC-based router. The method is highly adaptable to constrained hardware resources.

During the work on this thesis, especially during experimental evaluation of the method, the author identified issues and research topics that should be studied as the future work.

First and foremost, further insight has to be acquired into the relationship of FDD variable ordering and size of the resulting structure. As we have seen in the experiments (cf. Table 6.4), small changes in variable order can have enormous effects on the structure size. Such behaviour is typical for BDD-based structures. In general, finding an optimal BDD variable ordering is NP-complete. Heuristics are typically used to guess suitable variable ordering in BDDs. On the other hand, filtering diagrams are not as general as BDDs, so it is probable that a good method to determine optimal or near-to-optimal results, or at least a good heuristics could be found thanks to the limited domain of the problem.

In the experimental simulation environment presented in this work, ARP table updates are not supported. In order to evaluate the behaviour of the system in a more realistic environment, this feature should be included. When the target hardware architecture is finished, the overall performance of the whole accelerator will be also possible to validate.

The process of rewriting the RAF structure can adapt to limited memory sizes; the expansion into the associative memory can be stopped at any point preserving correctness of the result (cf. Section 5.8). We have discussed that such approach is possible. The problem how to choose parameters of the expansions in order to (1) obtain a feasible solution, i.e., fitting into the memories, (2) maximise CAM utilisation, and (3) maximise overall performance is left as a future work. Such method should adapt according to traffic characteristics in order to expand records that are used most frequently in the current traffic pattern. In order to recognise the traffic pattern, obtaining some feedback from the classification engine (i.e., usage counters) is necessary. Such approach turns the classification unit into a "cache with computation possibilities."

Another degree of freedom that can be used to optimise CAM utilisation is the possibility of choosing various widths of the CAM. Contemporary CAMs support trading width for number of records stored, typically in discrete steps. This property can be used as another parameter both in the process of variable order choice and adapting to memory size constraints.

The method as we described relies on recomputing the whole structure when source classification tables are changed. In order to minimise uploading the hardware memories, it is suitable to find possibilities to upgrade only altered parts of the memories when the structure gets changed.

Finally, the target architecture shall not be understood as given and fixed. Numerous improvements are possible in the hardware design. To give an example, the choice of variable classes does not have to be fixed for the whole CAM but it can depend on some properties of packets, e.g., protocol. Such changes may naturally modify our initial assumptions about the target architecture.

Bibliography

Al-Shaer, E. and Hamed, H. (2004). Discovery of policy anomalies in distributed firewalls. In *Proceedings of the 23rd Conference IEEE Communications Society (INFOCOM 2004)*.

Andersen, D. G., Balakrishnan, H., Kaashoek, M. F. and Morris, R. (2001). Resilient Overlay Networks. In *Symposium on Operating Systems Principles*, pages 131–145.

Andersen, H. R. (1998a). An Introduction to Binary Decision Diagrams. Department of Information Technology, Technical University of Denmark, Lyngby, http://www.it.dtu.dk/~hra/bdd97.ps.

Andersson, A. and Nilsson, S. (1993). Improved Behaviour of Tries by Adaptive Branching. *Information Processing Letters*, 46:295–300.

Andersson, A. and Nilsson, S. (1994). Faster Searching in Tries and Quadtrees—An Analysis of Level Compression. In *Proceedings of Second Annual European Symposium on Algorithms*, pages 82–93.

Andersson, A. and Nilsson, S. (1995). Efficient Implementation of Suffix Trees. *Software—Practice and Experience*, 25(2):129–141.

Antoš, D. (2001). PatLib, Pattern Manipulating Library. Master's thesis, Faculty of Informatics.

Antoš, D., Kořenek, J., Minaříková, K. and Řehák, V. (2003). Packet header matching in Combo6 IPv6 router. Technical Report, CESNET, z. s. p. o..

Antoš, D. and Kořenek, J. (2004). String Matching for IPv6 Routers. In Boas, P. V. E., Pokorný, J., Bielikova, M. and Štuller, J., editors, *SOFSEM 2004*, pages 205–210, Měřín, Czech Republic. MATFYZPRESS, Prague. ISBN 80-86732-19-3.

Antoš, D., Řehák, V. and Kořenek, J. (2004). Hardware Router's Lookup Machine and its Formal Verification. In *ICN'2004 Conference Proceedings*, pages 1002–1007, Gosier, Guadeloupe, French Caribbean. University of Haute Alsace, Colmar, France. ISBN 0-86341-325-0.

Attar, A. (2002). Performance Characteristics of BDD-Based Packet Filters. Technical Report, University of the Witwatersrand, Johannesburg.

Baboescu, F., Singh, S. and Varghese, G. (2003). Packet Classification for Core Routers: Is there an alternative to CAMs?. In *Proceedings of the IEEE INFOCOM*. IEEE Communications Society.

Baboescu, F. and Varghese, G. (2001). Scalable packet classification. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 199–210. SIGCOMM: ACM, ACM Press, New York, NY, USA. ISBN 1-58113-411-8.

Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D. and Macii, E. et al. (1993). Algebraic Decision Diagrams and Their Applications. In *IEEE/ACM International Conference on CAD*, pages 188–191, Santa Clara, California, USA. IEEE Computer Society Press.

Bollig, B. and Wegener, I. (1996). Improving the Variable Ordering of OBDDs Is NP-Complete. *IEEE Transactions on Computers*, 45(9):993–1002. ISSN 0018-9340.

Brace, K. S., Rudell, R. L. and Bryant, R. E. (1991). Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 40–45, Orlando, Florida, USA. IEEE/ACM, ACM Press, New York, NY, USA. ISBN 0-89791-363-9.

Bryant, R. E. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691. ISSN 0018-9340.

Bryant, R. E. (1992). Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318.

Bryant, R. E. (1995). Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification. In *IEEE/ACM International Conference on Computer Aided Design, ICCAD, San Jose, CA*, pages 236–243. IEEE CS Press, Los Alamitos.

Cheswick, W. R., Bellovin, S. M. and Rubin, A. D. (2003). *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley Professional, Second edition. ISBN 020163466X.

Cheung, G. and McCanne, S. (1999). Optimal Routing Table Design for IP Address Lookups Under Memory Constraints. In *INFOCOM (3)*, pages 1437–1444.

Christiansen, M. and Fleury, E. (2004). An Interval Decision Diagram Based Firewall. In *Proceedings of 3rd IEEE International Conference on Networking (ICN '04)*. University of Haute Alsace, Colmar, France. ISBN 0-86341-325-0.

Christiansen, M. and Fleury, E. (2004). An MTIDD Based Firewall: Using decision diagrams for packet filtering. *Telecommunications Systems*, 27(2–4):297–319.

Cisco Systems, I. (2005). Cisco IOS Software Configuration. http://www.cisco.com/univercd/cc/td/doc/product/software/.

Claffy, K. (1999). Internet measurement and data analysis: topology, workload, performance and routing statistics. http://traffic.caida.org/Reading/Papers/Nae/.

Compact Filter. (2005). Compact Filter: An IDD Based Packet Filter for Linux. http://www.cs.aau.dk/~mixxel/cf/.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2001). *Introduction to Algorithms*. MIT Press, Second edition. ISBN 0-262-03293-7.

Crescenzi, P., Dardini, L. and Grossi, R. (1999a). IP Address Lookup Made Fast and Simple. Technical Report, Universita di Pisa, Corso Italia 40, 56125 Pisa, Italy.

Eatherton, W. N. (1999). Hardware-Based Internet Protocol Prefix Lookups. Master's thesis, Washington University in St. Louis.

Eronen, P. and Zitting, J. (2001). An expert system for analyzing firewall rules. In *Proceedings of the 6th Nordic Workshop on Secure IT Systems*, pages 100–107.

Feamster, N., Andersen, D. G., Balakrishnan, H. and Kaashoek, M. F. (2003). Measuring the Effects of Internet Path Faults on Reactive Routing. In *Proc. of ACM SIGMETRICS 2003*.

Feldmann, A. and Muthukrishnan, S. (2000). Tradeoffs for Packet Classification. In *Proceedings of INFOCOM*, pages 1193–1202. IEEE.

Frantzen, L. (2003). Approaches for Analysing and Comparing Packet Filtering in Firewalls. Master's thesis, Technical University of Berlin.

FreeBSD. (2005). FreeBSD. http://www.freebsd.org.

Gupta, P. and McKeown, N. (1999). Packet classification on multiple fields. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 147–160, Cambridge, Massachusetts, United States. ACM Press, New York, NY, USA. ISBN 1-58113-135-6.

Gupta, P. and McKeown, N. (2000). Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41.

Hažmuk, I. (2005). Unified Header structure. Liberouter [Liberouter, 2005] CVS, file liberouter/vhdl_design/units/hfe/doc/UH-structure.txt.

Hazelhurst, S., Attar, A. and Sinnappan, R. (2000). Algorithms for Improving the Dependability of Firewall and Filter Rule Lists. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, pages 576–585, Washington, DC, USA. IEEE Computer Society. ISBN 0-7695-0707-7.

Hazelhurst, S., Fatti, A. and Henwood, A. (1998). Binary Decision Diagram Representations of Firewall and Router Access Lists. Technical Report, University of the Witwatersrand, Johannesburg, South Africa.

Höfer, F. (2003). Packet Analysis for IPv6 Router Implemented by a PCI Acceleration card. Technical Report, CESNET, z. s. p. o..

IPF. (2005a). IP Filter. http://coombs.anu.edu.au/~avalon/.

IPFIREWALL. (2005). Freebsd ipfirewall. FreeBSD ipfw(8) manual page.

Juniper Networks, Inc. (2005a). JUNOS Internet Software for J-series, M-series, and T-series Routing Platforms: Policy Framework Configuration Guide. http://www.juniper.net/techpubs/software/junos/junos71/index.html.

Karn, P. (2005). KA9Q routing package. http://www.ka9q.net/code/ka9qnos/.

Knuth, D. E. (1998). *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Second edition.

Kuhns, F., DeHart, J., Kantawala, A., Keller, R. and Lockwood, J. et al. (2002). Design of a high performance dynamically extensible router. In *Proceedings of DARPA Active Networks Conference and Exhibition*.

Kurose, J. E. and Ross, K. W. (2001). *Computer Networking—A Top-down Approach Featuring the Internet*. Addison-Wesley, Reading, Massachusetts, Second edition.

Lakshman, T. V. and Stiliadis, D. (1998). High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching. In *Proceedings of SIGCOMM '98*, pages 203–214. IEEE/ACM.

Liang, F. M. (1983). *Word Hy-phen-a-tion by Com-put-er*. PhD thesis, Department of Computer Science, Stanford University, Stanford, CA 94305.

Liberouter (2005). Liberouter Project WWW Page. http://www.liberouter.org.

Lidl, K. J., Lidl, D. G. and Borman, P. R. (2002). Flexible Packet Filtering: Providing a Rich Toolbox. In *Proceedings of the BSDCon '02 Conference on File and Storage Technologies*, pages 99–110, Cathedral Hill Hotel, San Francisco, California, USA. USENIX.

Lu, H. (2004). Improved Trie Partitioning for Cooler TCAMs. In *IASTED International Conference on Advances in Computer Science and Technology (ACST 2004)*.

Malkin, G. (1995). RFC 1868: ARP Extension—UNARP. Status: EXPERIMENTAL..

Mayer, A., Wool, A. and Ziskind, E. (2000). Fang: A Firewall Analysis Engine. In *IEEE Symposium on Security and Privacy*, pages 177–189, Berkeley, California.

Micheli, G. D., Ernst, R. and Wolf, W., editors (2002). *Readings in Hardware/Software Co-design* Morgan Kaufmann. ISBN 1-55860-702-1.

Minaříková, K. (2005). Computing Look-up Programs of Routing Accelerator. Master's thesis, Faculty of Informatics, Masaryk University Brno.

Minaříková, K. and Höfer, F. (2005). LUP Instruction Set. Liberouter [Liberouter, 2005] CVS, file liberouter/vhdl_design/units/lup/prog/nsim/instruction _m.def.

Moestedt, A. and Sjödin, P. (1998). IP Address Lookup in Hardware for High-Speed Routing. In *Proc. IEEE Hot Interconnects 6 Symposium*, pages 31–39, Stanford, California, USA.

Mogul, J. C. (1984). RFC 917: Internet subnets.

Mogul, J. C. and Postel, J. (1985). RFC 950: Internet Standard Subnetting Procedure. Updates RFC0792.

NetBSD. (2005). NetBSD. http://www.netbsd.org.

netfilter. (2005a). The netfilter/iptables project. http://www.netfilter.org/.

Nilsson, S. and Karlsson, G. (1999). IP-Address Lookup Using LC-Tries. In *IEEE Journal on Selected Areas in Communications*, pages 1083–1092.

Novotný, J., Fučík, O. and Antoš, D. (2003b). Liberouter—New Way in IPv6 Routers. In *ICETA 2003 2nd International Conference Proceedings*, pages 153–158. elfa, Košice, elfa, Košice.

Novotný, J., Fučík, O. and Antoš, D. (2003a). Project of IPv6 Router with FPGA Hardware Accelerator. In Cheung, P. Y., Constantinides, G. A. and de Sousa, J. T., editors, *Field-Programmable Logic and Applications, 13th International Conference FPL 2003*, pages 964–967. Springer Verlag.

Novotný, J., Fučík, O. and Kokotek, R. (2002). Schematics and PCB of COMBO6 card. Technical Report, CESNET, z. s. p. o..

OpenBSD. (2005). OpenBSD. http://www.openbsd.org.

Oppermann, A. (2006). TCP/IP Cleanup and Optimizations. http://people.freebsd.org/~andre/tcpoptimization.html; Performance improvements for the if_em driver: http://www.freebsd.org/cgi/cvsweb.cgi/src/sys/dev/em/if_em.c?rev=1.98&content-type=text/x-cvsweb-markup.

Pao, D., Liu, C., Wu, A., Yeung, L. and Chan, K. S. (2002). Efficient Hardware Architecture for Fast IP Address Lookup. In *IEEE INFOCOM 2002*, pages 555–561.

PF. (2005a). PF: The OpenBSD Packet Filter. http://www.openbsd.org/faq/pf/.

Plummer, D. C. (1982). RFC 826: Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware.

Rizzo, L. (2004a). New arp code snapshot for review. Mailing list FreeBSD-current, http://lists.freebsd.org/pipermail/freebsd-current/2004-April/026380.html.

Russel, R. (2005a). Linux IP Firewalling Chains. http://people.netfilter.org/~rusty/ipchains/.

Sahni, S. and Kim, K. S. (2003). Efficient Construction of Multibit Tries for IP Lookup. *IEEE/ACM Transactions on Networking (TON)*, 11(4):650–662. ISSN 1063-6692.

Sahni, S. and Kim, K. S. (2004). An $O(\log n)$ Dynamic Router-Table Design. *IEEE Transactions on Computers*, 53(3):351–363.

Singh, S., Baboescu, F., Varghese, G. and Wang, J. (2003). Packet Classification Using Multidimensional Cutting. In *SIGCOMM'03: Proceedings of the Applications, Technologies, Architectures, and Protocols for Computer Communication Conference*, pages 213–224, Karlsruhe, Germany. ACM Press, New York, NY, USA.

Sinnappan, R. and Hazelhurst, S. (2001). A Reconfigurable Approach to Packet Filtering. In Brebner, G. J. and Woods, R., editors, *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, number 2147 in LNCS, pages 638–642, Belfast, Northern Ireland, UK. Springer Verlag. ISBN 3-540-42499-7.

Sinnappan, R. A. (2001). A Reconfigurable Approach to TCP/IP Packet Filtering. Master's thesis, Faculty of Science, University of the Witwatersrand, Johannesburg.

Sklower, K. (1993). A tree-based routing table for Berkeley Unix. Technical Report, University of California, Berkeley.

Spitznagel, E., Taylor, D. and Turner, J. (2003). Packet Classification Using Extended TCAMs. In *Proceedings of ICNP*.

Srinivasan, V., Suri, S. and Varghese, G. (1999). Packet classification using tuple space search. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 135–146, New York, NY, USA. ACM Press. ISBN 1-58113-135-6.

Srinivasan, V. and Varghese, G. (1999). Fast Address Lookups using Controlled Prefix Expansion. *Transactions on Computer Systems*, 1(17):1–40.

Srinivasan, V., Varghese, G., Suri, S. and Waldvoge, M. (1998). Fast and Scalable Layer Four Switching. In *Proceedings of ACM SIGCOMM '98*, pages 191–202.

Strehl, K. and Thiele, L. (1998b). Symbolic model checking of process networks using interval diagram techniques. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 686–692, San Jose, California, United States. ACM Press, New York, NY, USA. ISBN 1-58113-008-2.

Taylor, D. E. (2004). Survey & Taxonomy of Packet Classification Techniques. Technical Report, Washington University in St. Louis.

Taylor, D. E., Lockwood, J. W., Sproull, T. S., Turner, J. S. and Parlour, D. B. (2002a). Scalable IP Lookup for Programmable Routers. In *IEEE INFOCOM 2002: 21st Annual Joint Conference of the IEEE Computer and Communications Societies.*

Taylor, D. E. and Spitznagel, E. W. (2005). On using content addressable memory for packet classification. Technical Report, Washington University in St. Louis.

Taylor, D. E. and Turner, J. S. (2005). Scalable Packet Classification using Distributed Crossproducting of Field Labels. In *IEEE INFOCOM 2005.*

Waldvogel, M. (2000). Multi-Dimensional Prefix Matching Using Line Search. In *Proceedings of IEEE Local Computer Networks,*, pages 200–207, Tampa, FL, USA.

Waldvogel, M., Varghese, G., Turner, J. and Plattner, B. (1997). Scalable High Speed IP Routing Lookups. In *Proceedings SIGCOMM 97.*

Waldvogel, M., Varghese, G., Turner, J. and Plattner, B. (2001). Scalable High-Speed Prefix Matching. *ACM Transactions on Computer Systems*, 19(4):440–482.

Wang, M. (2005). A Growth-Based Address Allocation Scheme for IPv6. In Boutaba, R., Almeroth, K. C., Puigjaner, R., Shen, S. X. and Black, J. P., editors, *Proceedings of Networking 2005*, number 3462 in Lecture Notes in Computer Science, pages 671–783. Springer Verlag. ISBN 3-540-25809-4.

Woo, T. Y. (2000). A Modular Approach to Packet Classification: Algorithms and Results. In *Proceedings of the IEEE INFOCOM*, pages 1213–1222, Tel Aviv, Israel. IEEE.

Zane, F., Narlikar, G. and Basu, A. (2003). CoolCAMs: Power-Efficient TCAMs for Forwarding Engines. In *Proceedings of IEEE INFOCOM 2003.*

A Structure of the Unified Header

Unified Header is a structure containing header fields on fixed positions. It serves to allow the lookup to abstract from actual structure of the packet. The structure has been proposed by Ivo Hažmuk, the complete description (including physical allocation of Unified Header registers) is available in CVS file [Hažmuk, 2005]. We present a brief overview here.

# of bits	description
16	L2 flag register (L2_REG, see below)
16	L3 flag register (L3_REG, see below)
48	DSTMAC—destination MAC address
48	SRCMAC—source MAC address
4	802.1p priority
12	802.1q VLAN tag
4	SRC iface ID (coded as a bitmap)
128	SRC IP address (IPv4 uses 32 LSB)
128	DST IP address (IPv4 uses 32 LSB)
16	SRC port/ICMP options (only for TCP/UDP/ICMP)
16	DST port (only for TCP/UDP)
128	INTER_ADDR Intermediate IP address (if Routing Header and/or IPv4 Source Route option is present)
16	PLEN—total packet length
8	PROTOCOL—Protocol number—/etc/services (aligned to 16 bits)
16	DRAM—allocation block number in DRAM (refers to the packet body)

L3_REG

bits	description
0	ARP—Ethernet Type 0x0806
1	TCP (for IPv4/IPv6) Ethernet (for ARP)
2	UDP (for IPv4/IPv6) IP (for ARP)
3	ICMP/ICMPv6 (for IPv4/IPv6) unused for ARP
4	Destination Options header for IPv6 IPv4 fragment unused for ARP

5	Encapsulation Security Protocol header (ESP) for IPv4/IPv6 unused for ARP
6	Authentication header (AH) present (for IPv4/IPv6) unused for ARP
7	Routing header for IPv6 unused for ARP and IPv4
8	Hop-by-Hop Options header for IPv6 unused for ARP and IPv4
9	INTER_ADDR is filled for IPv6 unused for ARP and IPv4
10	Hop Limit/TTL unused for ARP
11	nasty protocol/option (unrecognised, should be processed by the operating system) for IPv4/IPv6 unused for ARP
12	unknown protocol/option
13	bad packet
14	reserved
15	1—IPv6; 0—IPv4 unused for ARP

Meaning of most of fields of the L3 attribute register L3_REG depends on protocol type. Meaning of bits 1 and 2 depends on bit 0 (i.e., if the packet is ARP). Meaning of bits 4–10 depends on IP version denoted by bit 15.

L2_REG

bits	description
0	CRC 0—good, 1—bad
1	length 0—OK, 1—longer than MTU
2	RUNT 0—OK, 1—shorter than 64 bytes
3	SRCMAC belonging to the interface (0)
4	unknown Ethernet protocol
5	802.3 (0) or other protocol (1)
6	BAD_DSTMAC: 0—DSTMAC is ready, 1—bad recipient ditto
7	NOTMYMAC: 0—Unicast packet is for me, 1—another recipient
8	OTHER_ERR: 0—good, 1—other unspecified error
9	802.1Q
10	MPLS unicast
11–15	reserved

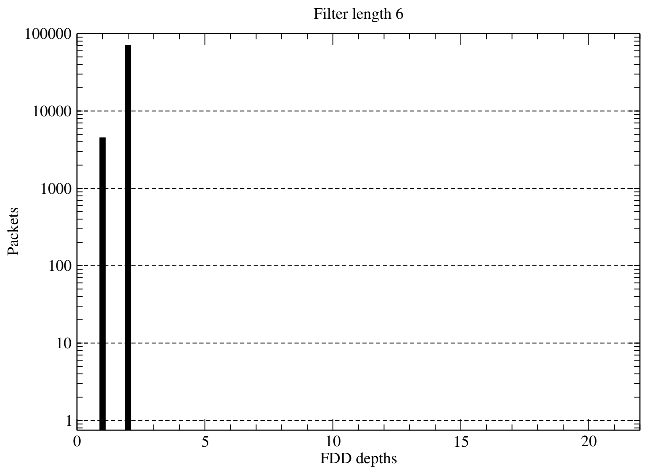
B Detailed Experimental Results

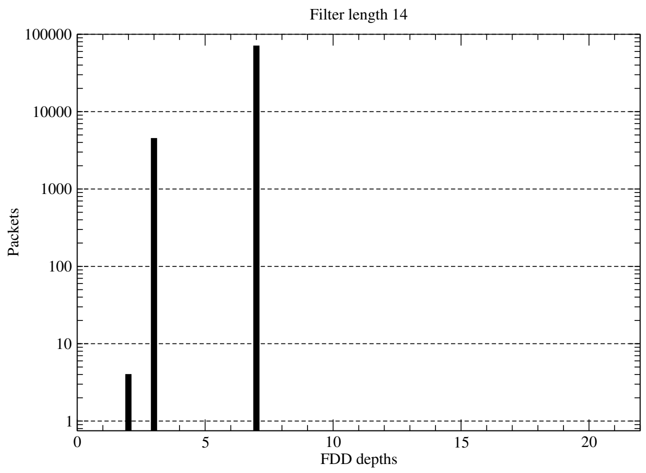
This appendix covers detailed results of measurements presented in Chapter 6.

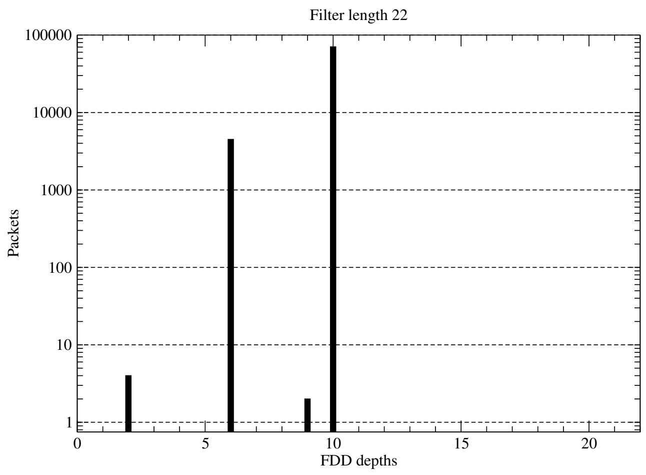
B.1 Variable Filter Length

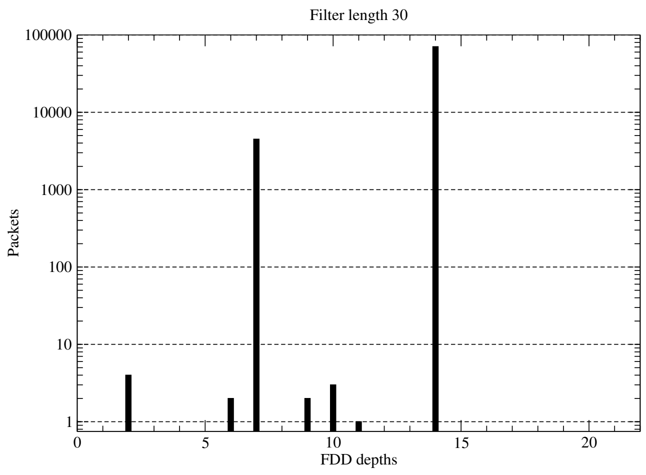
See Section 6.4.1.4 for explanation of the results presented here.

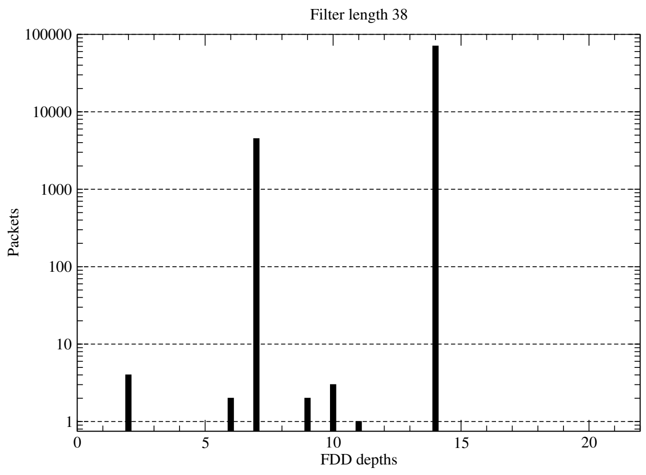
B.1.1 Data Set 1

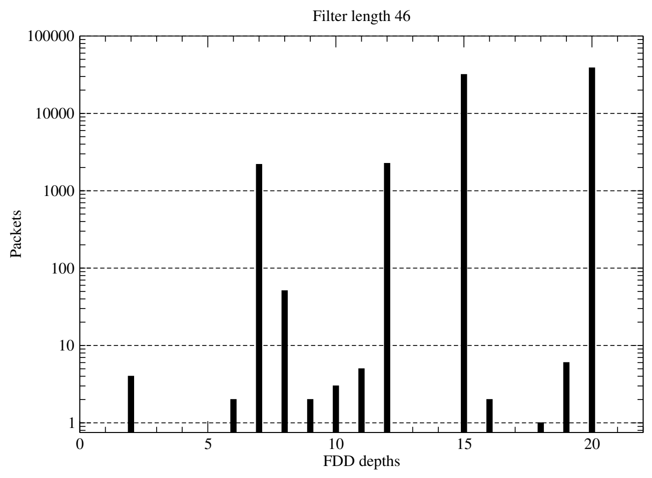


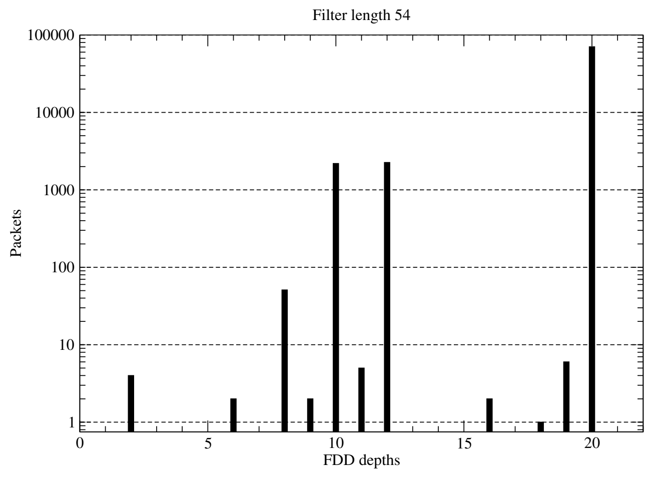


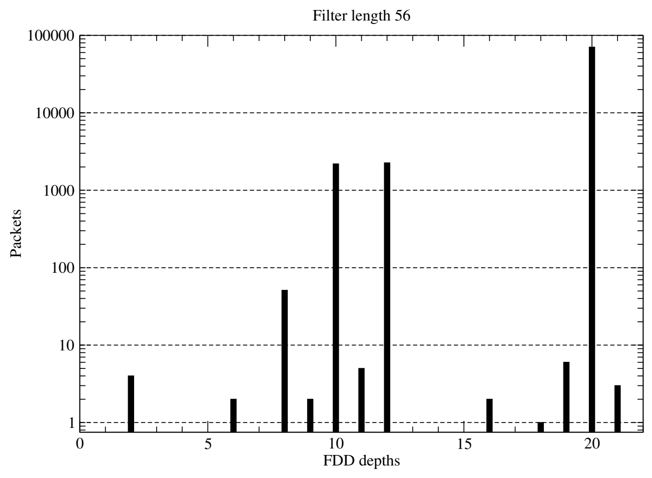

Filter length 22
100000
10000
1000
100
10
1
Packets
0
5
10
15
20
FDD depths


Filter length 30
100000
10000
1000
100
10
1
Packets
0
5
10
15
20
FDD depths

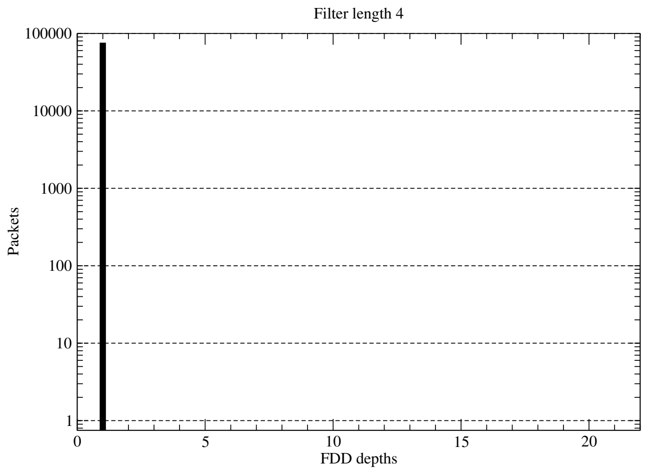

Filter length 38
100000
10000
1000
100
10
1
Packets
0
5
10
15
20
FDD depths

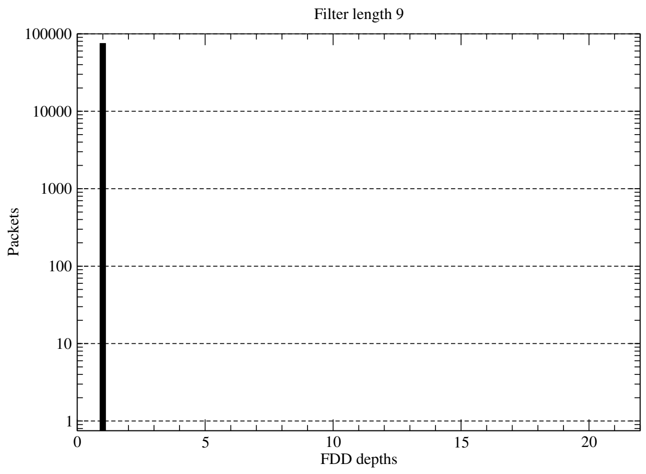

Filter length 46
Packets
FDD depths
1
10
100
1000
10000
100000
0
5
10
15
20

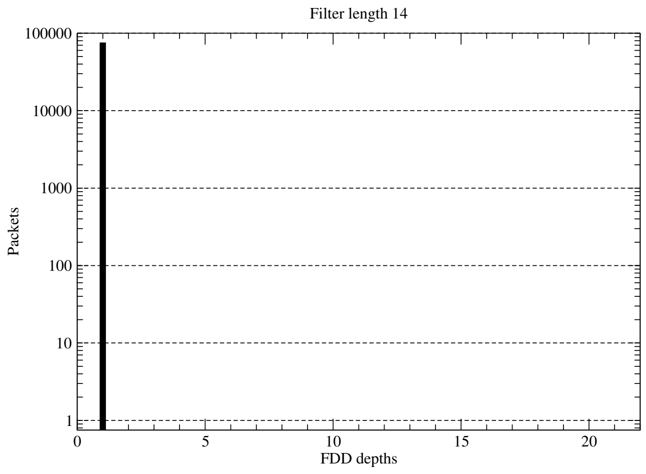

Filter length 54
Packets
FDD depths
1
10
100
1000
10000
100000
0
5
10
15
20

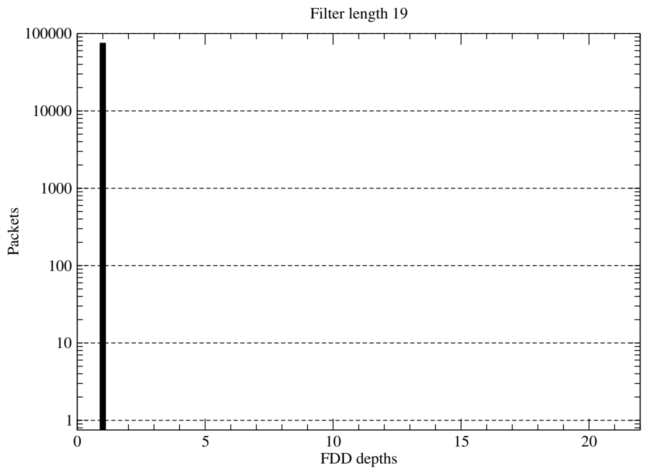

Filter length 56
Packets
FDD depths
1
10
100
1000
10000
100000
0
5
10
15
20

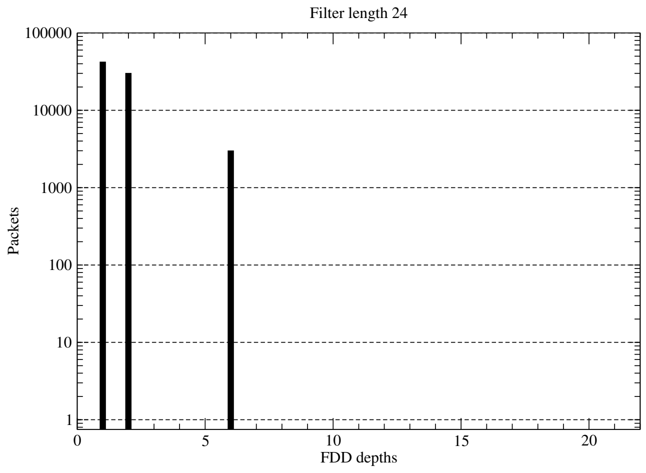
B.1.2 Data Set 2

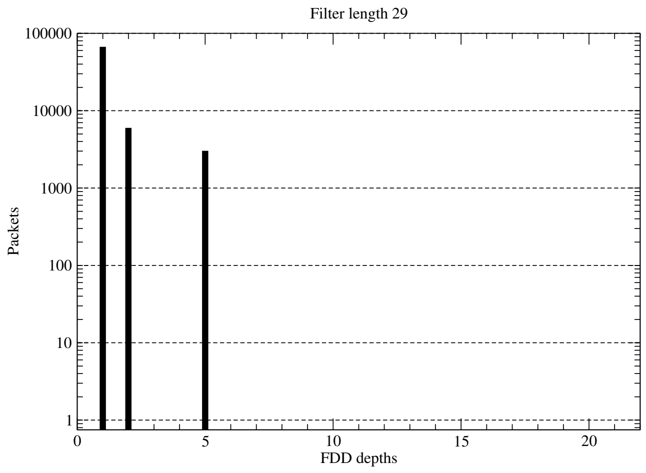


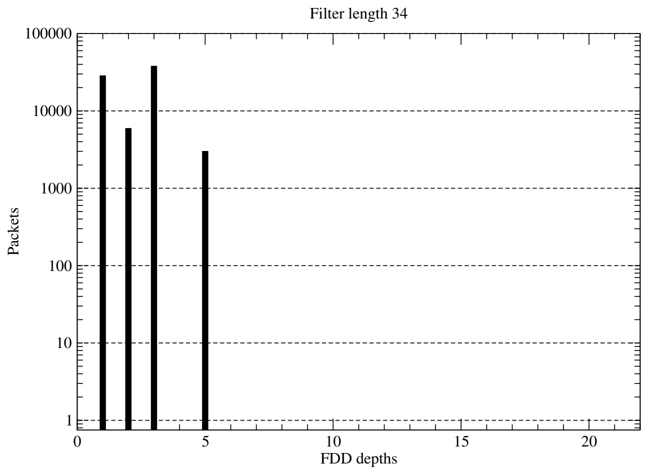


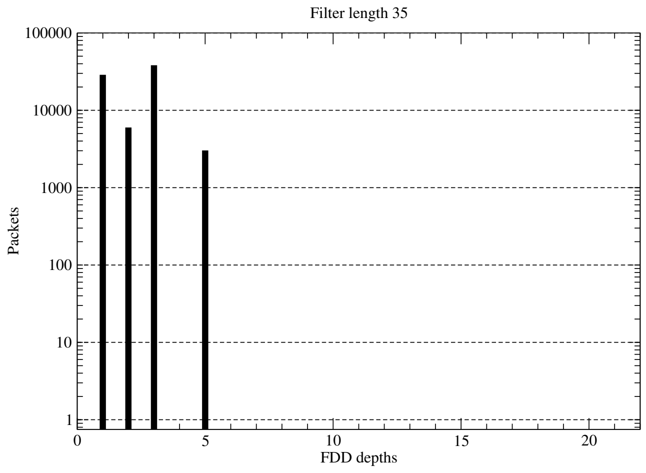



Filter length 19
Packets
FDD depths
100000
10000
1000
100
10
1
0
5
10
15
20

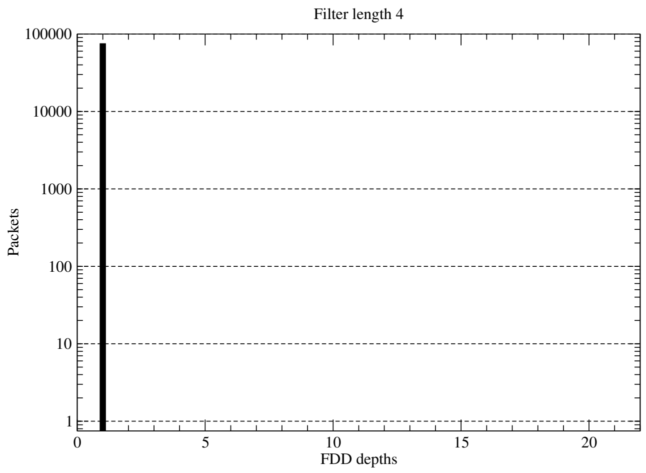

Filter length 24
Packets
FDD depths
100000
10000
1000
100
10
1
0
5
10
15
20

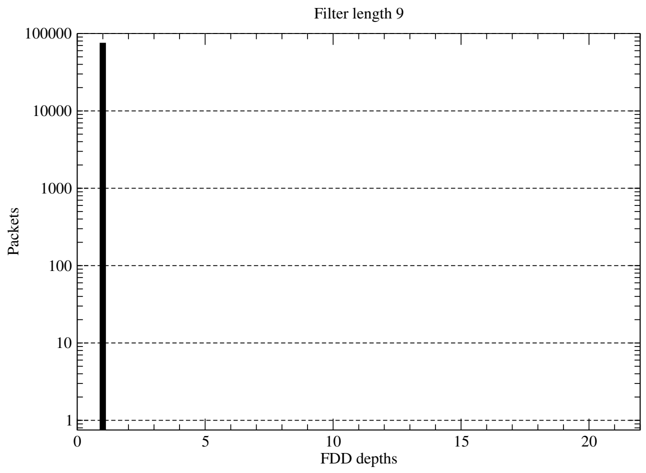

Filter length 29
Packets
FDD depths
100000
10000
1000
100
10
1
0
5
10
15
20

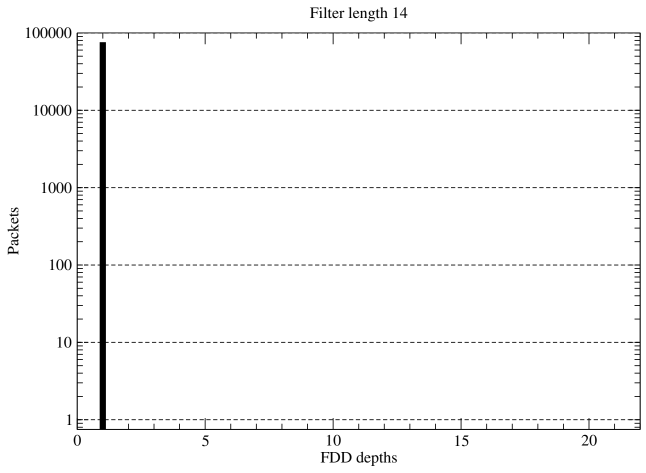


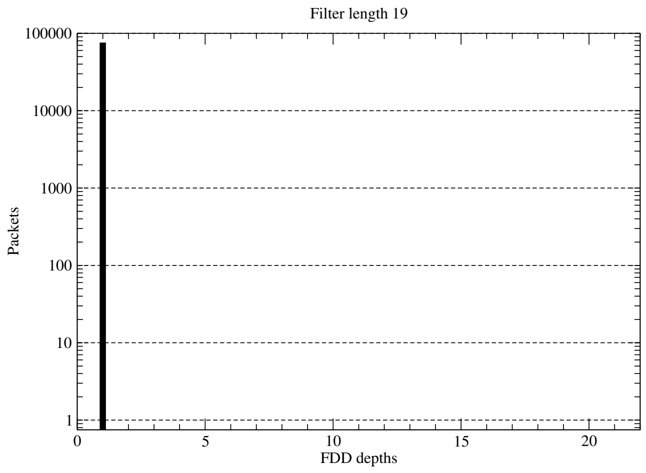


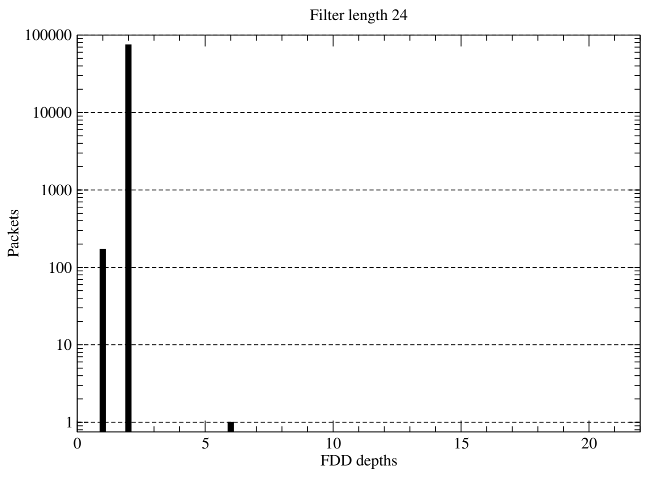
B.1.3 Data Set 3

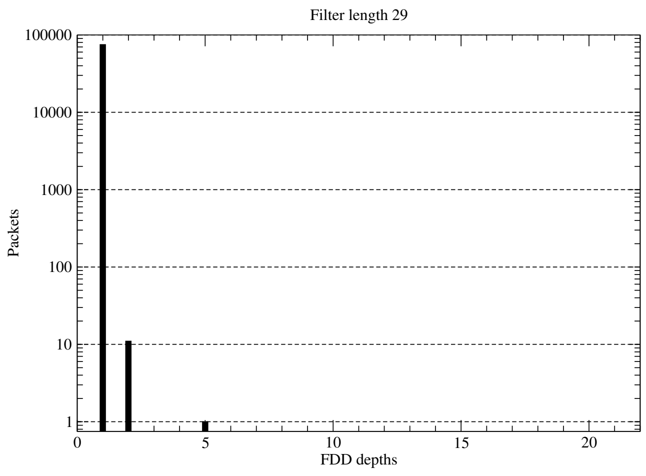


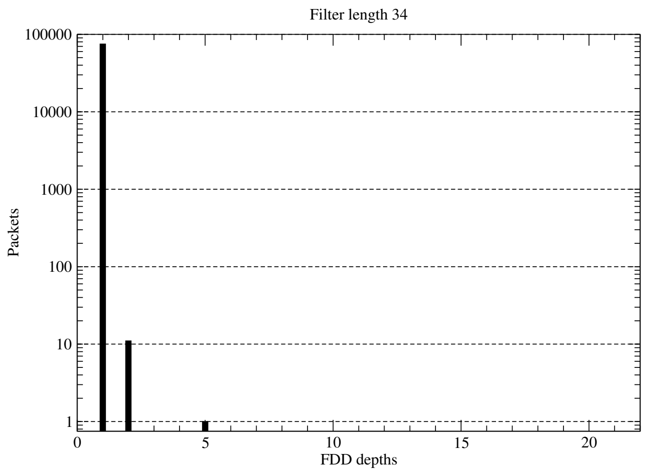

Filter length 9
Packets
FDD depths
100000
10000
1000
100
10
1
0
5
10
15
20

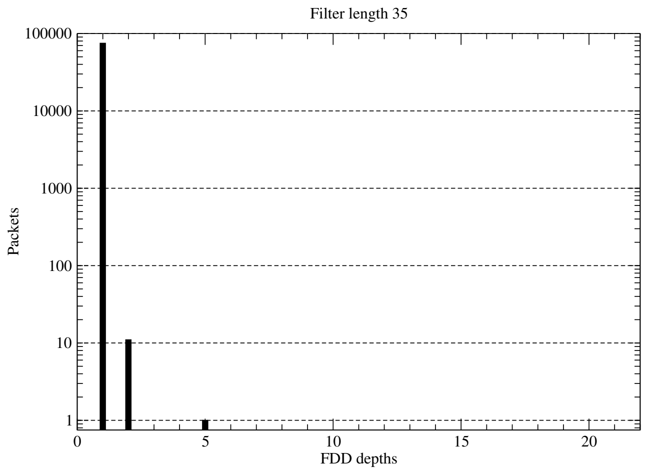

Filter length 14
Packets
FDD depths
100000
10000
1000
100
10
1
0
5
10
15
20


Filter length 19
Packets
FDD depths
100000
10000
1000
100
10
1
0
5
10
15
20


Filter length 24
100000
10000
1000
100
10
1
Packets
0
5
10
15
20
FDD depths


Filter length 29
100000
10000
1000
100
10
1
Packets
0
5
10
15
20
FDD depths

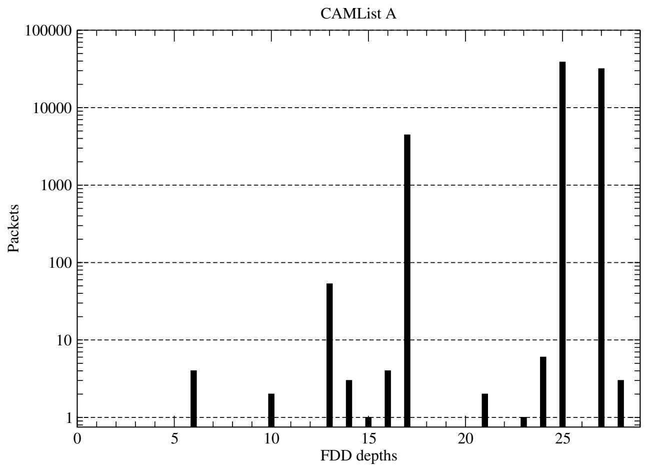

Filter length 34
100000
10000
1000
100
10
1
Packets
0
5
10
15
20
FDD depths

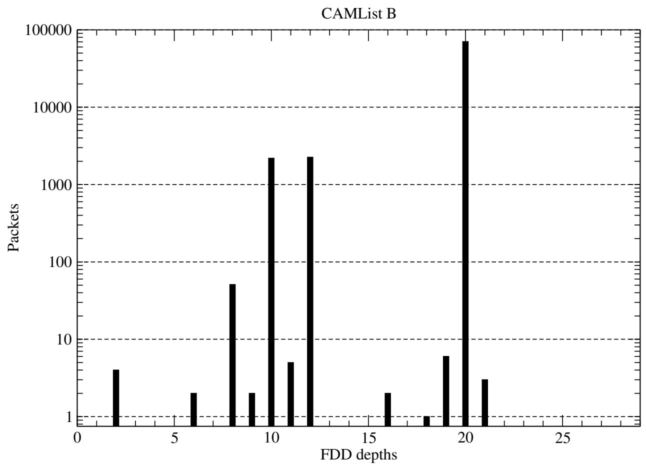


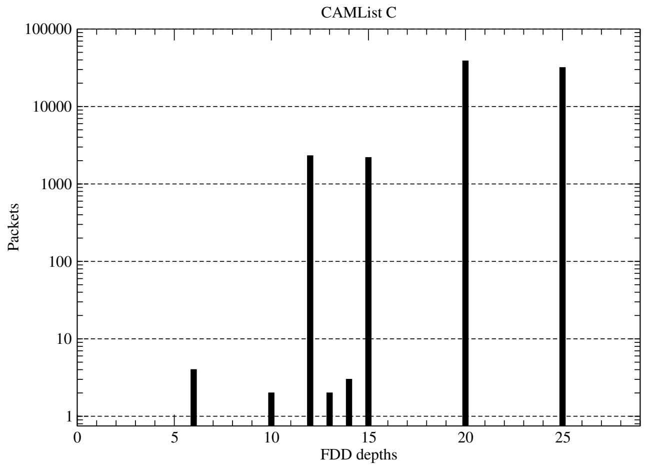
B.2 Effects of *CAMList*

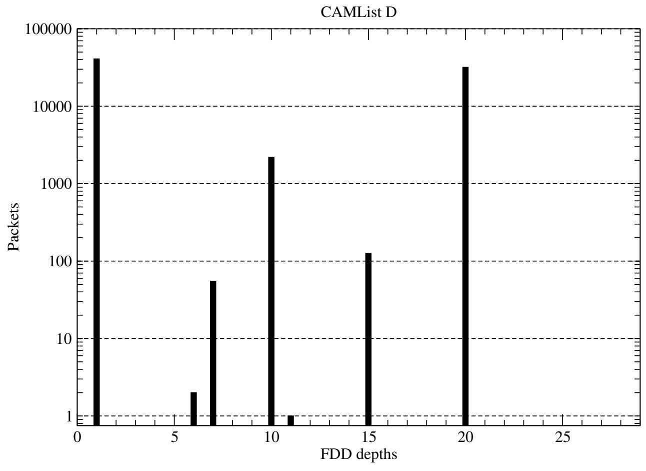
See Section 6.4.3 for explanation of results presented here.

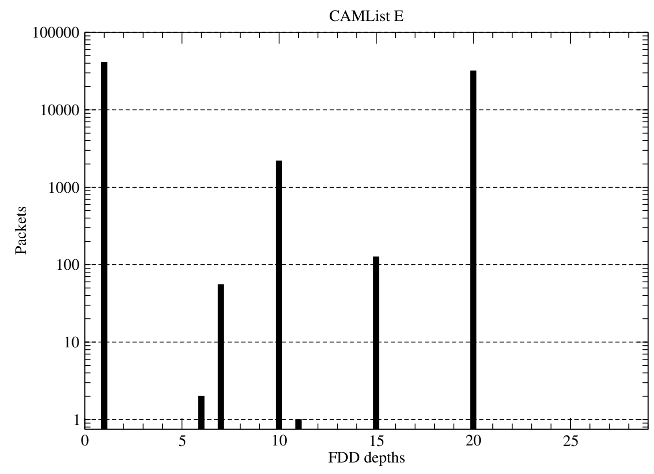
B.2.1 Data Set 1



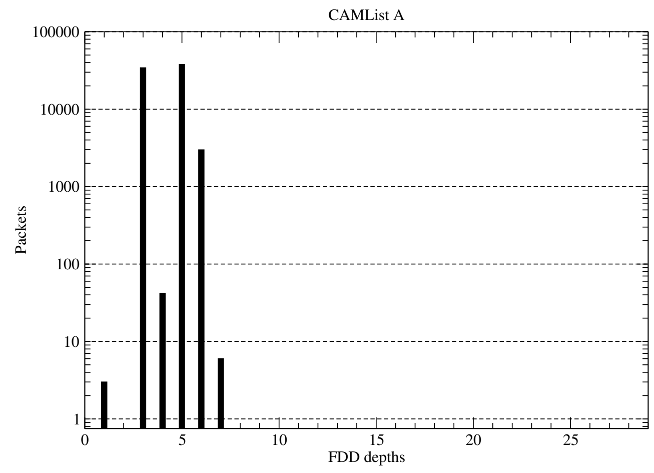

CAMList B
Packets
FDD depths
100000
10000
1000
100
10
1
0
5
10
15
20
25

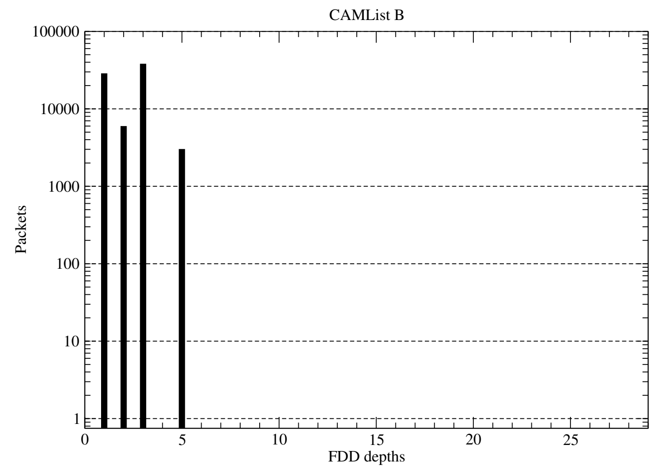

CAMList C
Packets
FDD depths
100000
10000
1000
100
10
1
0
5
10
15
20
25

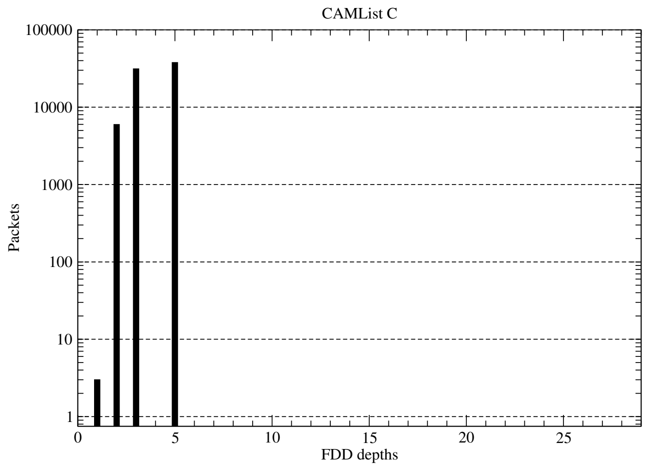

CAMList D
Packets
FDD depths
100000
10000
1000
100
10
1
0
5
10
15
20
25

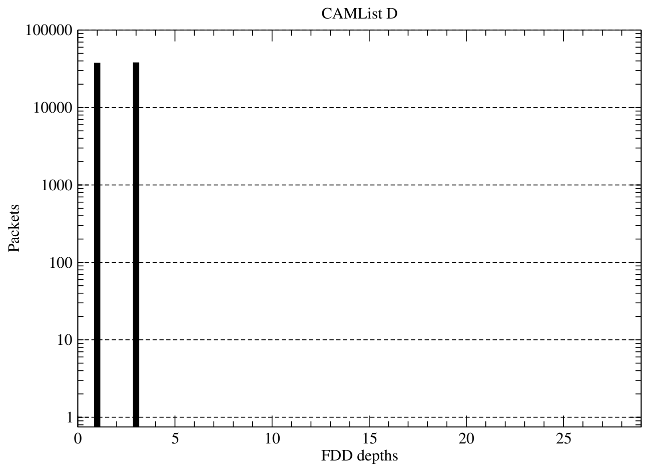


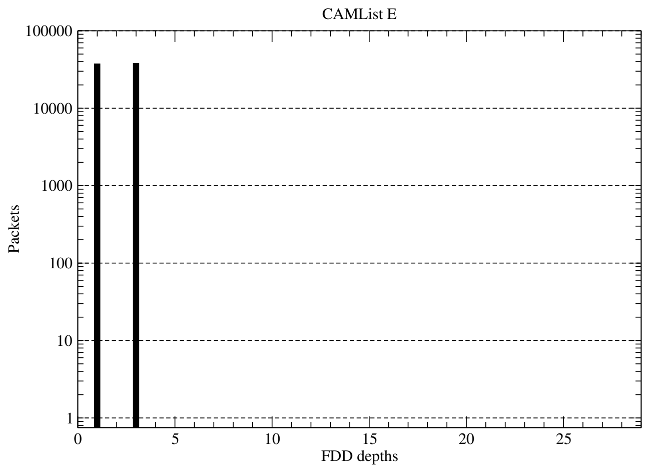
B.2.2 Data Set 2



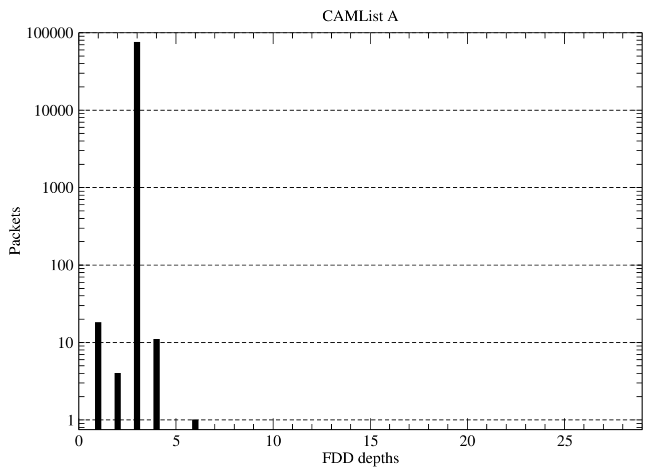


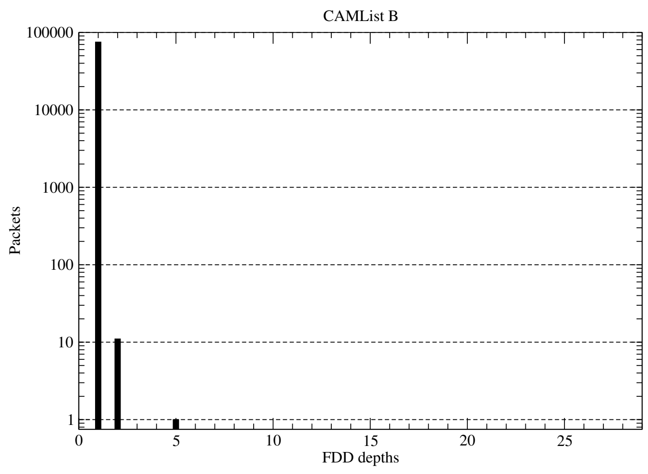

CAMList C
100000
10000
1000
100
10
1
Packets
0
5
10
15
20
25
FDD depths

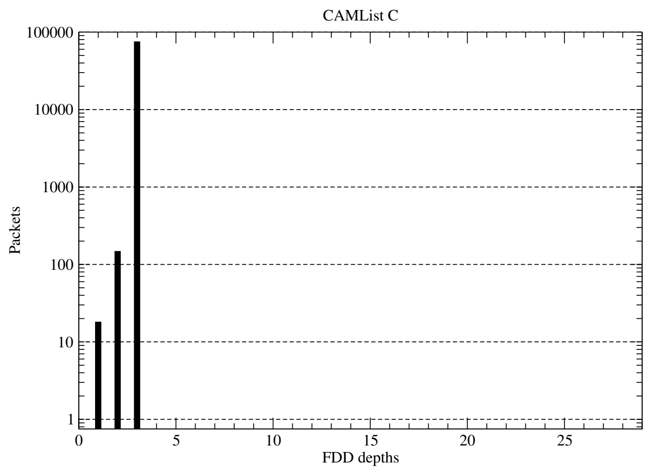

CAMList D
100000
10000
1000
100
10
1
Packets
0
5
10
15
20
25
FDD depths

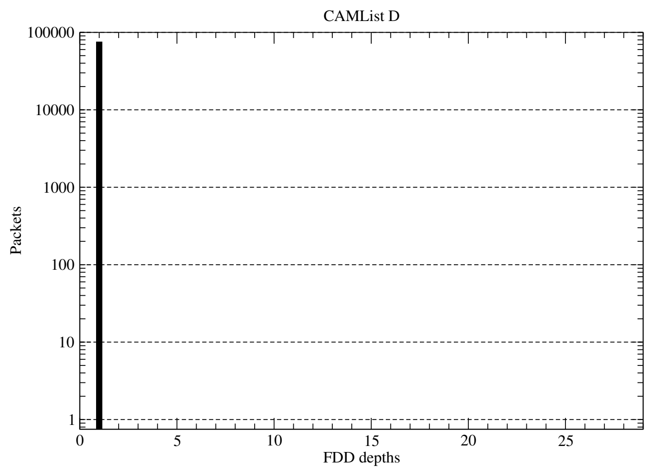

CAMList E
100000
10000
1000
100
10
1
Packets
0
5
10
15
20
25
FDD depths

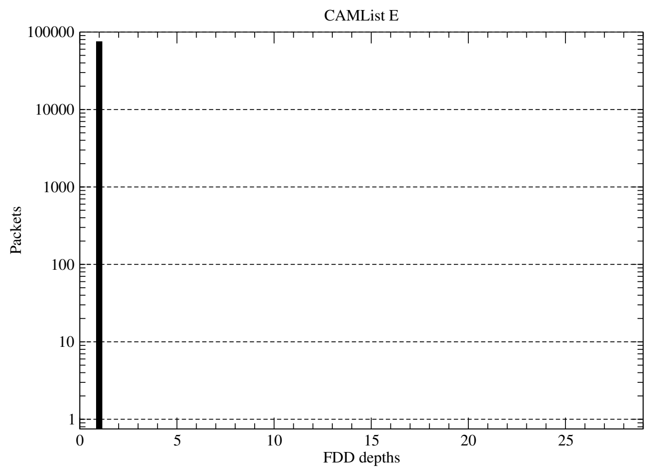
B.2.3 Data Set 3








CAMList D
Packets
FDD depths
100000
10000
1000
100
10
1
0
5
10
15
20
25


CAMList E
Packets
FDD depths
100000
10000
1000
100
10
1
0
5
10
15
20
25