Report Number: WUCSE-2004-28

2004-05-21

ClassBench: A Packet Classification Benchmark

David E. Taylor and Jonathan S. Turner

Due to the importance and complexity of the packet classification problem, a myriad of algorithms and re-sulting implementations exist. The performance and capacity of many algorithms and classification devices, including TCAMs, depend upon properties of the filter set and query patterns. Unlike microprocessors in the field of computer architecture, there are no standard performance evaluation tools or techniques avail-able to evaluate packet classification algorithms and products. Network service providers are reluctant to distribute copies of real filter sets for security and confidentiality reasons, hence realistic test vectors are a scarce commodity. The small subset of the research community who... **Read complete abstract on page 2.**

ClassBench: A Packet Classification Benchmark

David E. Taylor and Jonathan S. Turner

Complete Abstract:

Due to the importance and complexity of the packet classification problem, a myriad of algorithms and resulting implementations exist. The performance and capacity of many algorithms and classification devices, including TCAMs, depend upon properties of the filter set and query patterns. Unlike microprocessors in the field of computer architecture, there are no standard performance evaluation tools or techniques avail-able to evaluate packet classification algorithms and products. Network service providers are reluctant to distribute copies of real filter sets for security and confidentiality reasons, hence realistic test vectors are a scarce commodity. The small subset of the research community who obtain real filter sets either limit performance evaluation to the small sample space or employ ad hoc methods of modifying those filter sets. In response to this problem, we present ClassBench, a suite of tools for benchmarking packet classification algorithms and devices. ClassBench includes a Filter Set Generator that produces synthetic filter sets that accurately model the characteristics of real filter sets. Along with varying the size of the filter sets, we provide high-level control over the composition of the filters in the resulting filter set. The tools suite also includes a Trace Generator that produces a sequence of packet headers to exercise the synthetic filter set. Along with specifying the relative size of the trace, we provide a simple mechanism for controlling locality of reference in the trace. While we have already found ClassBench to be very useful in our own research, we seek to initiate a broader discussion and solicit input from the community to guide the refinement of the tools and codification of a formal benchmarking methodology.

2004-28

ClassBench: A Packet Classification Benchmark

Authors: Taylor, David E.; Turner, Jonathan S.

May 21, 2004

Abstract: Due to the importance and complexity of the packet classification problem, a myriad of algorithms and resulting
implementations exist. The performance and capacity of many algorithms and classification devices, including TCAMs, depend upon properties of the filter set and query patterns. Unlike microprocessors in the field of computer architecture, there are no standard performance evaluation tools or techniques available to evaluate packet classification algorithms and products. Network service providers are reluctant to distribute copies of real filter sets for security and confidentiality reasons, hence realistic test vectors are a scarce commodity. The small subset of the research community who obtain real filter sets either limit performance evaluation to the small sample space or employ ad hoc methods of modifying those filter sets. In response to this problem, we present ClassBench, a suite of tools for benchmarking packet classification algorithms and devices. ClassBench includes a Filter Set Generator that produces synthetic filter sets that accurately model the characteristics of real filter sets. Along with varying the size of the filter sets, we provide high-level control over the composition of the filters in the resulting filter set. The tools suite also includes a Trace Generator that produces a sequence of packet headers to exercise the synthetic filter set. Along with specifying the relative size of the trace, we provide a simple mechanism for controlling locality of reference in the trace. While we have already found ClassBench to be very useful in our own research

ClassBench: A Packet Classification Benchmark

David E. Taylor, Jonathan S. Turner

WUCSE-2004-28

May 21, 2004

Applied Research Laboratory
Department of Computer Science and Engineering
Washington University in Saint Louis
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130
davidtaylor@wustl.edu

Abstract

Due to the importance and complexity of the packet classification problem, a myriad of algorithms and resulting implementations exist. The performance and capacity of many algorithms and classification devices, including TCAMs, depend upon properties of the filter set and query patterns. Unlike microprocessors in the field of computer architecture, there are no standard performance evaluation tools or techniques available to evaluate packet classification algorithms and products. Network service providers are reluctant to distribute copies of real filter sets for security and confidentiality reasons, hence realistic test vectors are a scarce commodity. The small subset of the research community who obtain real filter sets either limit performance evaluation to the small sample space or employ ad hoc methods of modifying those filter sets. In response to this problem, we present *ClassBench*, a suite of tools for benchmarking packet classification algorithms and devices. *ClassBench* includes a *Filter Set Generator* that produces synthetic filter sets that accurately model the characteristics of real filter sets. Along with varying the size of the filter sets, we provide high-level control over the composition of the filters in the resulting filter set. The tools suite also includes a *Trace Generator* that produces a sequence of packet headers to exercise the synthetic filter set. Along with specifying the relative size of the trace, we provide a simple mechanism for controlling locality of reference in the trace. While we have already found *ClassBench* to be very useful in our own research, we seek to initiate a broader discussion and solicit input from the community to guide the refinement of the tools and codification of a formal benchmarking methodology.

1 Introduction

Deployment of next generation network services hinges on the ability of Internet infrastructure to provide flow identification at physical link speeds. A packet classifier must compare header fields of every incoming packet against a set of filters in order to identify a flow. The resulting flow identifier is used to apply security policies, application processing, and quality-of-service guarantees to packets belonging to the specified flow. Typical packet classification filter sets have fewer than a thousand filters and reside in enterprise firewalls or edge routers. As network services continue to migrate into the network core, it is anticipated that filter sets could swell to tens of thousands of filters or more. The most common type of packet classification examines the packet headers fields comprising the standard IP 5-tuple: IP source and destination address, transport protocol number, and source and destination port numbers. A packet classifier searches for the highest priority filter or set of filters matching the packet where each filter specifies a prefix on the IP addresses, an exact match or wildcard on the transport protocol number, and ranges on the transport port numbers. As we discuss in Section 3.6, packet filters often specify fields beyond the standard IP 5-tuple and we anticipate that filter sets will continue to scale to larger numbers of fields. For this reason, we designed *ClassBench* with the capability of generating filter sets with additional filter fields such as TCP flags and ICMP type numbers. While this is an important feature, the primary contribution of our work is the accurate modeling of the structure of the filter fields comprising the standard IP 5-tuple.

As reported in Section 3, it has been observed that real filter sets exhibit a considerable amount of structure. In response, several algorithmic techniques have been developed which exploit filter set structure to accelerate search time or reduce storage requirements [1, 2, 3, 4]. Consequently, the performance of these approaches are subject to the structure or statistical characteristics of the filter set. Likewise, the capacity and efficiency of the most prominent packet classification solution, Ternary Content Addressable Memory (TCAM), is also subject to the characteristics of the filter set [5, 6]. Despite the influence of filter set composition on the performance of packet classification search techniques and devices, no publicly available benchmarking tools, filter sets, or formal methodology exists for standardized performance evaluation. Due to security and confidentiality issues, access to large, real filter sets for analysis and performance measurements of new classification techniques has been limited to a small subset of the research community. Some researchers in academia have gained access to filter sets through confidentiality agreements, but are unable to distribute those filter sets. Furthermore, performance evaluations using real filter sets are restricted by the size and structure of the sample filter sets.

In order to facilitate future research and provide a foundation for a meaningful benchmark, we present *ClassBench*, a publicly available suite of tools for benchmarking packet classification algorithms and devices. As shown in Figure 1, *ClassBench* consists of three tools: a *Filter Set Analyzer*, *Filter Set Generator*, and *Trace Generator*. The general approach of *ClassBench* is to construct a set of benchmark *parameter files* that specify the relevant characteristics of real filter sets, generate a synthetic filter set from a chosen *parameter file* and small set of high-level inputs, and also provide the option to generate a sequence of packet headers to probe the synthetic filter set using the *Trace Generator*. *Parameter files* contain various statistics and probability distributions that guide the generation of synthetic filter sets. The *Filter Set* analyzer tool extracts the relevant statistics and probability distributions from a seed filter set and generates a *parameter file*. This provides the capability to generate large synthetic filter sets which model the structure of a seed filter set. In Section 4 we discuss the statistics and probability distributions contained in the *parameter files* that drive the synthetic filter generation process. Selection of the relevant statistics and distributions is based on our study of 12 real filter sets presented in Section 3, and several iterations of the *Filter Set Generator* design. Note that *parameter files* may also be hand-constructed from qualitative characterizations of a specific filter set or class of filter sets such as backbone routers, edge routers, etc. We envision a set of benchmark *parameter files* that may be refined or expanded over time as the tools enjoy broader use.

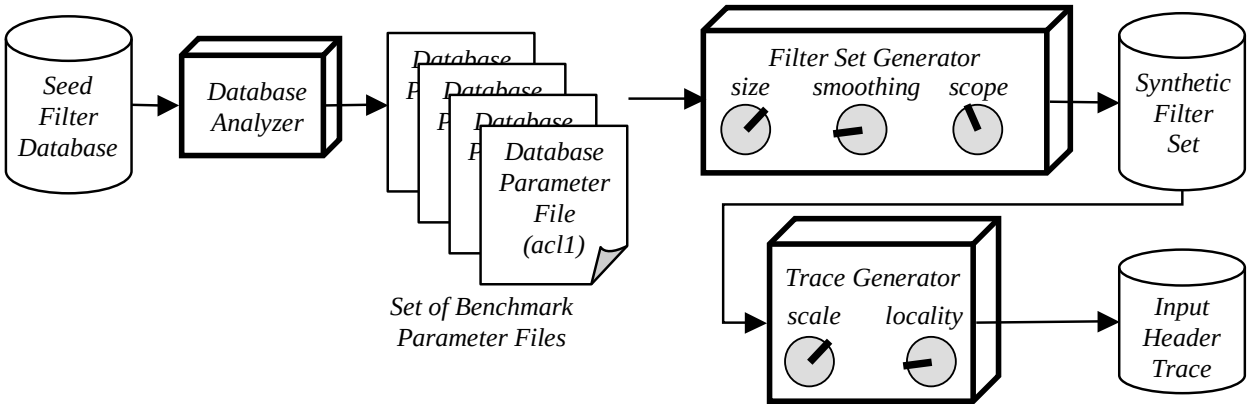


Figure 1: Block diagram of the *ClassBench* tools suite. The synthetic *Filter Set Generator* has size, smoothing, and scope adjustments which provide high-level, systematic mechanisms for altering the size and composition of synthetic filter sets. The set of benchmark *parameter files* model real filter sets and may be refined over time. The *Trace Generator* provides adjustments for trace size and locality of reference.

The *Filter Set Generator* takes as input a *parameter file* and a few high-level parameters. Along with specifying filter set size, the tool provides mechanisms for systematically altering the composition of filters. Two adjustments, *smoothing* and *scope*, provide high-level control over filter set generation and an abstraction from the low-level statistics and distributions contained in the *parameter files*. The *smoothing* adjustment provides a structured mechanism for introducing new address aggregates which is useful for modeling filter sets significantly larger than the filter set used to generate the *parameter file*. The *scope* adjustment provides a biasing mechanism to favor more or less specific filters during the generation process. These adjustments and their affects on the resulting filter sets are discussed in Section 5.1 and Section 5.2. Finally, the *Trace Generator* tool examines the synthetic filter set, then generates a sequence of packet headers to exercise the filter set. Like the *Filter Set Generator*, the trace generator provides adjustments for scaling the size of the trace as well as the locality of reference of headers in the trace. These adjustments are described in detail in Section 6.

We highlight previous performance evaluation efforts by the research community as well as related benchmarking activity of the IETF in Section 2. It is our hope that this work initiates a broader discussion which will lead to refinement of the tools, compilation of a standard set of *parameter files*, and codification of a formal benchmarking methodology. Its value will depend on its perceived clarity and usefulness to the interested community. In the case of packet classification, this community is comprised of at least the following groups:

- *Researchers* seeking to evaluate new classification algorithms relative to alternative approaches and commercial products.
- *Classification product vendors* seeking to market their products with convincing performance claims over competing products.
- *Classification product customers* seeking to verify and compare classification product performance on a uniform scale. This group can be sub-divided into two major sub-groups: router vendors seeking to compare competing classification products during the design process and prior to selecting components, and router customers seeking to independently verify performance claims of router vendors based on the components used in the router.

2 Related Work

Extensive work has been done in developing benchmarks for many applications and data processing devices. Benchmarks are used extensively in the field of computer architecture to evaluate microprocessor performance. The effectiveness of these benchmarks to accurately distinguish the effects of architectural improvements, fabrication advances, and compiler optimizations is debatable; yet, there exists inherent value in providing a uniform scale for comparison.

In the field of computer communications, the Internet Engineering Task Force (IETF) has several working groups exploring network performance measurement. Specifically, the IP Performance Metrics (IPPM) working group was formed with the purpose of developing standard metrics for Internet data delivery [7]. The Benchmarking Methodology Working Group (BMWG) seeks to make measurement recommendations for various internetworking technologies [8][9]. These recommendations address metrics and performance characteristics as well as collection methodologies.

The BMWG specifically attacked the problem of measuring the performance of Forwarding Information Base (FIB) routers [10][11]. Realizing that router throughput, latency, and frame loss rate depend on the structure of the Forwarding Information Base (FIB) or route table, the BMWG prescribes a testing methodology with accompanying terminology. The recommendations describe testing at the router level, compounding the effects of system interfaces, control, and switching fabric. While the suggested tests take into consideration table size and prefix distribution, they lack specificity in how prefix distributions should be varied. The recommendations do introduce a methodology for determining the maximum FIB size and evaluating throughput relative to the table size. The BMWG also produced a methodology for benchmarking firewalls [12][13]. The methodology contains broad specifications such as: the firewall should contain at least one rule for each host, tests should be run with various filter set sizes, and test traffic should correspond to rules at the "end" of the filter set. This final specification provides for more accurate performance assessment of firewalls employing simple linear search algorithms. We assert that *ClassBench* complements efforts by the IETF by providing the necessary tools for generating test vectors with high-level control over filter set and input trace composition. The Network Processor Forum (NPF) has also initiated a benchmarking effort [14]. Currently, the NPF has produced benchmarks for switch fabrics and route lookup engines. To our knowledge, there are no current efforts by the IETF or the NPF to provide a benchmark for multiple field filter matching.

In the absence of publicly available packet filter sets, researchers have exerted much effort in order to generate realistic performance tests for new algorithms. Several research groups obtained access to real filter sets through confidentiality agreements. Gupta and McKeown obtained access to 40 real filter sets and extracted a number of useful statistics which have been widely cited [1]. Gupta and McKeown also generated synthetic two-dimensional filter sets consisting of source-destination address prefix pairs by randomly selecting address prefixes from publicly available route tables [2]. This technique was also employed by Feldman and Muthukrishnan [15]. Warkhede, Suri, and Varghese used this technique in a study of packet classification for two-dimensional "conflict-free" filters [16]. Baboescu and Varghese also generated synthetic two-dimensional filter sets by randomly selecting prefixes from publicly available route tables, but added refinements for controlling the number of zero-length prefixes (wildcards) and prefix nesting [3, 17]. A simple technique for appending randomly selected port ranges and protocols from real filter sets in order to generate synthetic five-dimensional filter sets is also described [3]. Baboescu and Varghese also introduced a simple scheme for using a sample filter set to generate a larger synthetic five-dimensional filter set [4]. This technique replicates filters by changing the IP prefixes while keeping the other fields unchanged. While these techniques address some aspects of scaling filter sets in size, they lack high-level mechanisms for adjusting the generation of new filter types and the specificity of filters.

Woo provided strong motivation for a packet classification benchmark and initiated the effort by providing an overview of filter characteristics for different environments (ISP Peering Router, ISP Core Router, Enterprise Edge Router, etc.) [18]. Based on high-level characteristics, Woo generated large synthetic filter sets, but provided few details about how the filter sets were constructed. The technique also does not provide controls for varying the composition of filters within the filter set. Nonetheless, his efforts provide a good starting point for constructing a benchmark capable of modeling various application environments for packet classification. Sahasranaman and Buddhikot used the characteristics compiled by Woo in a comparative evaluation of a few packet classification techniques [19].

3 Analysis of Real Filter Sets

Recent efforts to identify better packet classification techniques have focused on leveraging the characteristics of real filter sets for faster searches. While lower bounds for the general multi-field searching problem have been established, observations made in recent packet classification work offer enticing new possibilities to provide significantly better average performance. In this chapter, we summarize the observations made in the literature and report the results of our additional analyses. The focus of this section is to identify and understand the impetus for the observed structure of filter databases and to develop metrics and characterizations of filter set structure that aid in generating synthetic filter sets.

We performed a battery of analyses on 12 real filter sets provided by Internet Service Providers (ISPs), a network equipment vendor, and other researchers working in the field. The filter sets range in size from 68 to 4557 entries and utilize one of the following formats:

- Access Control List (ACL) - standard format for security, VPN, and NAT filters for firewalls and routers (enterprise, edge, and backbone)
- Firewall (FW) - proprietary format for specifying security filters for firewalls
- IP Chain (IPC) - decision tree format for security, VPN, and NAT filters for software-based systems

Due to confidentiality concerns, the filter sets were provided without supporting information regarding the types of systems and environments in which they are used. We are unable to comment on "where" in the network architecture the filter sets are used: enterprise core routers, ISP edge routers, backbone core routers, enterprise firewalls, etc. Nonetheless, the following analysis provide invaluable insight into the structure of real filter sets. We observe that various useful properties hold regardless of filter set size or format.

3.1 Understanding Filter Composition

Many of the observed characteristics of filter sets arise due to the administrative policies that drive their construction. The most complex packet filters typically appear in firewall and edge router filter sets due to the heterogeneous set of applications supported in these environments. Firewalls and edge routers typically implement security filters and network address translation (NAT), and they may support additional applications such as Virtual Private Networks (VPNs) and resource reservation. Typically, these filter sets are created manually by a system administrator using a standard management tool such as CiscoWorks VPN/Security Management Solution (VMS) [20] and Lucent Security Management Server (LSMS) [21]. Such tools conform to a model of filter construction which views a filter as specifying the communicating subnets and the application or set of applications. Hence, we can view each filter as having two major components: an

address prefix pair and an application specification. The address prefix pair identifies the communicating subnets by specifying a source address prefix and a destination address prefix. The application specification identifies a specific application session by specifying the transport protocol, source port number, and destination port number. A set of applications may be identified by specifying ranges for the source and destination port numbers.

3.2 Previous Observations

Gupta and McKeown published a number of observations regarding the characteristics of real filter sets which have been widely cited [1]. Others have performed analyses on real filter sets and published their observations [4, 3, 22, 18, 23]. The following is a distillation of observations relevant to our discussion:

- Current filter set sizes are small, ranging from tens of filters to less than 5000 filters. It is unclear if the size limitation is "natural" or a result of the limited performance and high expense of existing packet classification solutions.
- The protocol field is restricted to a small set of values. In most filter sets, TCP, UDP, and the wildcard are the most common specifications; other specifications include ICMP, IGMP, (E)IGRP, GRE and IPINIP.
- Transport-layer specifications vary widely. There are common range specifications for port numbers such as 'gt 1023' (greater than 1023).
- The number of unique address prefixes matching a given address is typically five or less.
- Most prefixes have either a length of 0 or 32; there are some prefixes with lengths of 21, 23, 24 and 30.
- The number of filters matching a given packet is typically five or less.

Kounavis, et. al. performed a thorough analysis of four ACLs and proposed a general framework for packet classification in network processors [23]. The authors made a number of interesting observations and assertions. Specifically, they observed a dependency between the size of the ACL and the number of filters that have a wildcard in the source or destination IP address. The authors refer to filters that contain a wildcard in either the source or destination address as "partially specified". They found that partially specified filters comprise a smaller proportion of the filter set as the number of filters increases. Specifically, 83% of the filters in the smallest ACL were partially specified while only 10% of the filters in the largest ACL were partially specified. The authors also observed trends in the composition of partially specified filters. The smallest ACL from an enterprise firewall had large numbers of partially specified filters with destination address wildcards, while the largest ACL from an ISP had large numbers of partially specified filters with source address wildcards. The authors suggest that these characteristics are a result of the "location" of the ACL in the Internet. Small ACLs are "closer" to client subnets, therefore filters are used to block traffic flows from a number of specific subnets. Large ACLs are "closer" to the Internet backbone, thus filters are used to control access to a number of important servers or networks. The authors found that the number of filters matching a packet is typically four with a maximum of seven. Finally, they found that the number of unique application specifications (combination of transport protocol and port ranges) is small due to the limited number of popular applications in the Internet.

3.3 Application Specifications

We analyzed the application specifications in the 12 filter sets in order to corroborate previous observations as well as extract new, potentially useful characteristics.

3.3.1 Protocol

For each of the filter sets, we examined the unique protocol specifications and the distribution of filters over the set of unique values. As shown in Table 1, filters specified one of nine protocols or the wildcard. Note that two filter sets, fw2 and fw4, contain anonymized protocol numbers; therefore, we did not include them in our analysis. We observed the following protocol specifications, listed in order of frequency of occurrence:

- Transmission Control Protocol (TCP), RFC793 [24]
- User Datagram Protocol (UDP), RFC768 [25]
- Wildcard
- Internet Control Message Protocol (ICMP), RFC792 [26]
- General Routing Encapsulation (GRE), RFC2784 [27]
- Open Shortest Path First (OSPF) Interior Gateway Protocol (IGP), RFC2178 [28]
- Enhanced Interior Gateway Routing Protocol (EIGRP), Cisco [29]
- IP Encapsulating Security Payload (ESP) for IPv6, RFC2406 [30]
- IP Authentication Header (AH) for IPv6, RFC2402 [31]
- IP Encapsulation within IP (IPE), RFC2003 [32]

Like previous studies, the most common protocol specification is TCP. On average, TCP is specified by twice as many filters as the next most common protocol, UDP. The wildcard is the third most common specification. All filter sets contain a small number of filters specifying ICMP. The remaining six protocols are only specified by a few filters in a few of the filter sets.

3.3.2 Port Ranges

Next, we examined the port ranges specified by filters in the filter sets and the distribution of filters over the unique values. In order to observe trends among the various filter sets, we define five classes of port ranges:

- WC, wildcard
- HI, ephemeral user port range $[1023 : 65535]$
- LO, well-known system port range $[0 : 1023]$
- AR, arbitrary range
- EM, exact match

Table 1: Observed protocols and filter distribution; values given as percentage (%) of filters in the filter set.

Set	$*$	*ICMP*	*IPE*	*TCP*	*UDP*	*GRE*	*ESP*	*AH*	*EIGRP*	*OSPF IGP*
acl1	8.46	3.14	0.00	87.31	1.09	0.00	0.00	0.00	0.00	0.00
acl2	46.39	0.96	0.00	44.94	6.74	0.00	0.00	0.00	0.96	0.00
acl3	4.92	4.17	0.00	65.00	25.87	0.00	0.00	0.00	0.00	0.04
acl4	4.08	3.99	0.10	65.76	25.87	0.16	0.00	0.00	0.00	0.03
acl5	0.00	28.59	0.00	28.22	41.78	0.00	0.00	0.00	0.00	1.40
fw1	1.06	3.89	0.00	57.24	32.16	5.65	0.00	0.00	0.00	0.00
fw3	1.63	5.98	0.00	55.98	36.41	0.00	0.00	0.00	0.00	0.00
fw5	1.88	6.87	0.00	51.88	39.38	0.00	0.00	0.00	0.00	0.00
ipc1	34.49	1.12	0.00	26.15	37.72	0.29	0.12	0.12	0.00	0.00
ipc2	27.08	36.46	0.00	10.42	26.04	0.00	0.00	0.00	0.00	0.00
AVG	**13.00**	**9.52**	**0.01**	**49.29**	**27.31**	**0.61**	**0.01**	**0.01**	**0.10**	**0.15**

Motivated by the allocation of port numbers, the first three classes represent common specifications for a port range. The last two classes may be viewed as partitioning the remaining specifications based on whether or not an exact port number is specified. Table 2 shows the distribution of filters over the five port classes for both source and destination ports. We observe some interesting trends in the data. With rare exception, the filters in the ACL filter sets specify the wildcard for the source port. A majority of filters in the ACL filters specify an exact port number for the destination port. Source port specifications in the other filter sets are also dominated by the wildcard, but a considerable portion of the filters specify an exact port number. Destination port specifications in the other filter sets share the same trend, however the distribution between the wildcard and exact match is a bit more even. After the wildcard and exact match, the HI port class is the most common specification. A small portion of the filters specify an arbitrary range, 4% on average and at most 12%. Only one filter set contained filters specifying the LO port class for either the source or destination port range.

3.3.3 Port Pair Class

As previously discussed, the structure of source and destination port range pairs is a key point of interest for both modeling real filter sets and designing efficient search algorithms. We can characterize this structure by defining a *Port Pair Class* (PPC) for every combination of source and destination port class. For example, WC-WC if both source and destination port ranges specify the wildcard, AR-LO if the source port range specifies an arbitrary range and the destination port range specifies the set of well-known system ports. As shown in Figure 2, a convenient way to visualize the structure of *Port Pair Classes* is to define a *Port Pair Class* matrix where rows share the same source port class and columns share the same destination port class. For each filter set, we examine the PPC defined by filters specifying the same protocol. For all protocols except TCP and UDP, the PPC is trivial – a single spike at WC/WC. Figure 2 highlights the uniqueness of PPC matrices among different protocols and filter sets.

3.4 Address Prefix Pairs

A filter identifies communicating hosts or subnets by specifying a source and destination address prefix, or address prefix pair. The speed and efficiency of several longest prefix matching and packet classifi-

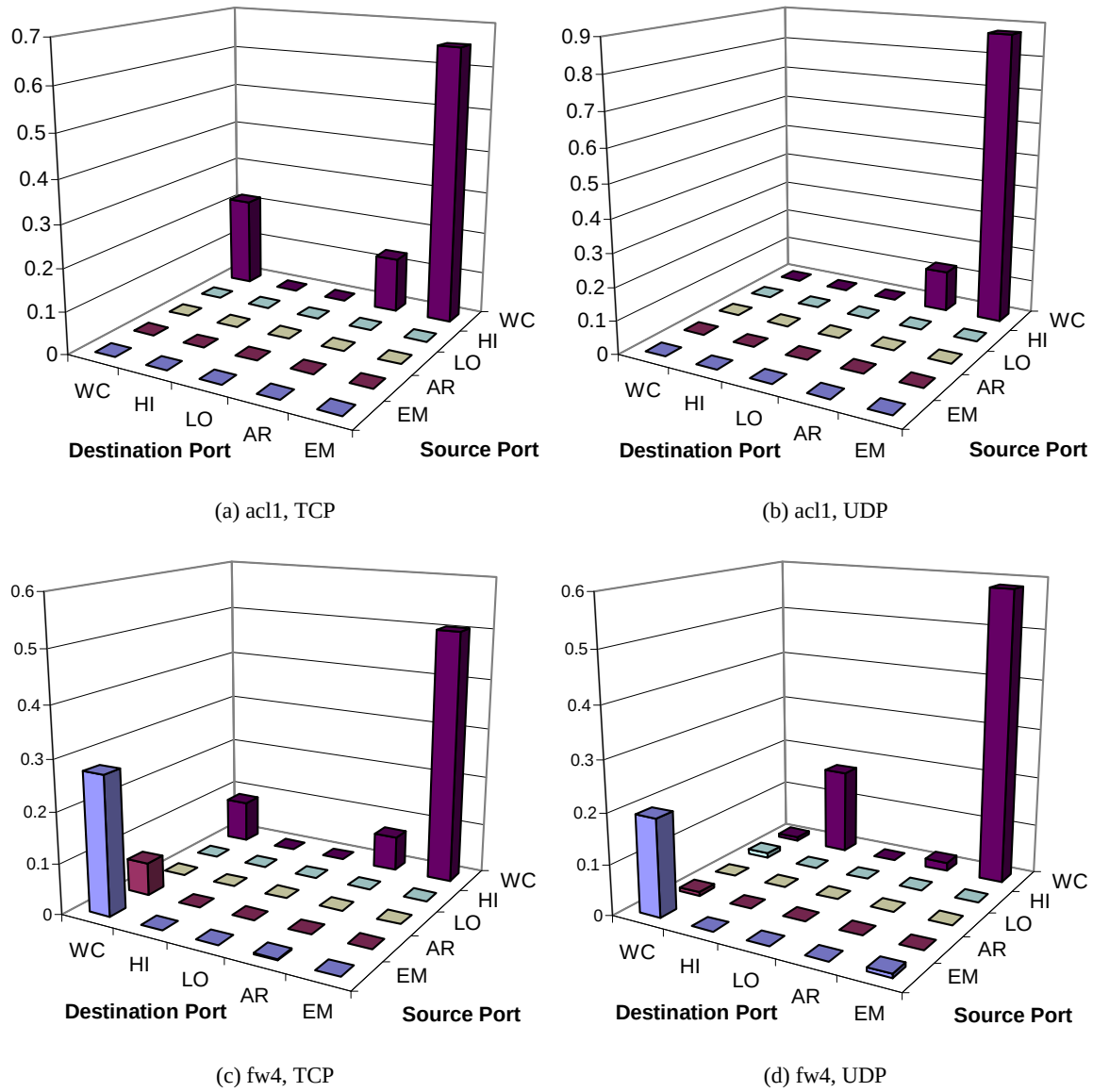
Table 2: Distribution of filters over the five port classes for source and destination port range specifications; values given as percentage (%) of filters in the filter set.

Set	*Source Port*					*Destination Port*				
	WC	*HI*	*LO*	*AR*	*EM*	*WC*	*HI*	*LO*	*AR*	*EM*
acl1	100.0	0.00	0.00	0.00	0.00	30.42	0.00	0.00	11.60	57.98
acl2	100.0	0.00	0.00	0.00	0.00	69.34	0.64	0.00	7.06	22.95
acl3	99.92	0.00	0.00	0.00	0.08	9.25	13.96	0.00	11.04	65.75
acl4	99.93	0.00	0.00	0.00	0.07	8.56	12.15	0.00	11.21	68.08
acl5	100.0	0.00	0.00	0.00	0.00	30.00	4.08	0.00	5.20	60.72
fw1	77.74	8.13	0.00	0.35	13.78	31.10	8.13	0.00	0.35	60.42
fw2	38.24	17.65	0.00	0.00	44.12	100.0	0.00	0.00	0.00	0.00
fw3	77.72	5.98	0.00	0.54	15.76	27.72	5.98	0.00	0.54	65.76
fw4	10.98	42.05	10.98	1.52	34.47	13.26	18.94	0.76	1.14	65.91
fw5	75.62	5.00	0.00	0.62	18.75	35.62	3.75	0.00	1.25	59.38
ipc1	82.84	0.35	0.00	2.00	14.81	55.46	6.52	0.00	2.53	35.49
ipc2	73.96	0.00	0.00	0.00	26.04	73.96	0.00	0.00	0.00	26.04
AVG	**78.08**	**6.60**	**0.92**	**0.42**	**13.99**	**40.39**	**6.18**	**0.06**	**4.33**	**49.04**

cation algorithms depend upon the number of unique prefix lengths and the distribution of filters across those unique values. We begin our analysis by examining the number of unique prefix lengths. In Table 3 we report the number of unique source address prefix lengths, destinations address prefix lengths, and source/destination prefix pair lengths for the 12 filter sets. A majority of the filter sets have more unique source address prefix lengths than unique destination prefix lengths. For all of the filter sets, the number of unique source/destination prefix pair lengths is small relative to the filter set size and the number of possible combinations, 1024.

Next, we examine the distribution of filters over the unique address prefix pair lengths. Note that this study is unique in that previous studies and models of filter sets utilized independent distributions for source and destination address prefixes. When constructing synthetic filter sets to test new packet classification algorithms, researchers often randomly select address prefixes from backbone route tables which are dominated by class C address prefixes (24-bit network address) and aggregates of class A, B, and C address prefixes. As shown in Figure 3, real filter sets have unique prefix pair distributions that reflect the types of filters contained in the filter set. For example, fully specified source and destination addresses dominate the distribution for acl5 shown in Figure 3(a). There are very few filters specifying a 24-bit prefix for either the source or destination address. Also consider the distribution for fw1 shown in Figure 3(c). The most common prefix pair is a fully specified destination address and a wildcard for the source address. This is due to the nature of the filter set, a firewall limiting access to a key host. It is not possible to model the prefix pair distribution using independent prefix length distributions, even if those distributions are taken from real filter sets. Finally, we observe that while the distributions are sufficiently different from each other a majority of the filters in the filter sets specify prefix pair lengths around the "edges" of the distribution. Note that there are not large "spikes" in or around the centers of the distributions in Figure 3. This implies that, typically, one of the address prefixes is either fully specified or wildcarded.

By considering the prefix pair distribution, we characterize the *size* of the communicating subnets specified by filters in the filter set. Next, we would like to characterize the relationships among address prefixes and the amount of address space covered by the prefixes in the filter set. Our primary motivation is to devise metrics to facilitate construction of synthetic filter sets that accurately model real filter sets. Consider a



(a) acl1, TCP

(b) acl1, UDP

(c) fw4, TCP

(d) fw4, UDP

Figure 2: Port Pair Matrices for two filter sets.

binary tree constructed from the IP source address prefixes of all filters in the filter set. From this tree, we could completely characterize the data structure by determining a branching probability for each node. For example, assume that an address prefix is generated by traversing the tree starting at the root node. At each node, the decision to take to the 0 path or the 1 path exiting the node depends upon the branching probability at the node. For a complete characterization of the tree, the branching probability at each node is unique. As shown in Figure 4, $p1|10$ is the probability that the 1 path is chosen at level 2 given that the 1 path was chosen at level 0 and the 0 path was chosen at level 1.

Such a characterization is infeasible, hence we employ suitable metrics that capture the important characteristics while providing a convenient abstraction. We begin by constructing two binary tries from the source and destination prefixes in the filter set. Note that there is one level in the tree for each possible prefix length 0 through 32 for a total of 33 levels. For each level in the tree, we compute the probability that a node

Table 3: Number of unique address prefix lengths for source address (SA), destination address (DA), and source/destination address pairs (SA/DA).

Set	*Size*	*SA*	*DA*	*SA/DA*
acl1	733	6	20	31
acl2	623	13	13	50
acl3	2400	22	12	89
acl4	3061	22	15	98
acl5	4557	11	3	31
fw1	283	12	6	22
fw2	68	4	3	8
fw3	184	9	3	13
fw4	264	5	6	12
fw5	160	10	4	17
ipc1	1702	15	13	93
ipc2	192	4	2	5

has one child or two children. Nodes with no children are excluded from the calculation. We refer to this distribution as the *Branching Probability*.

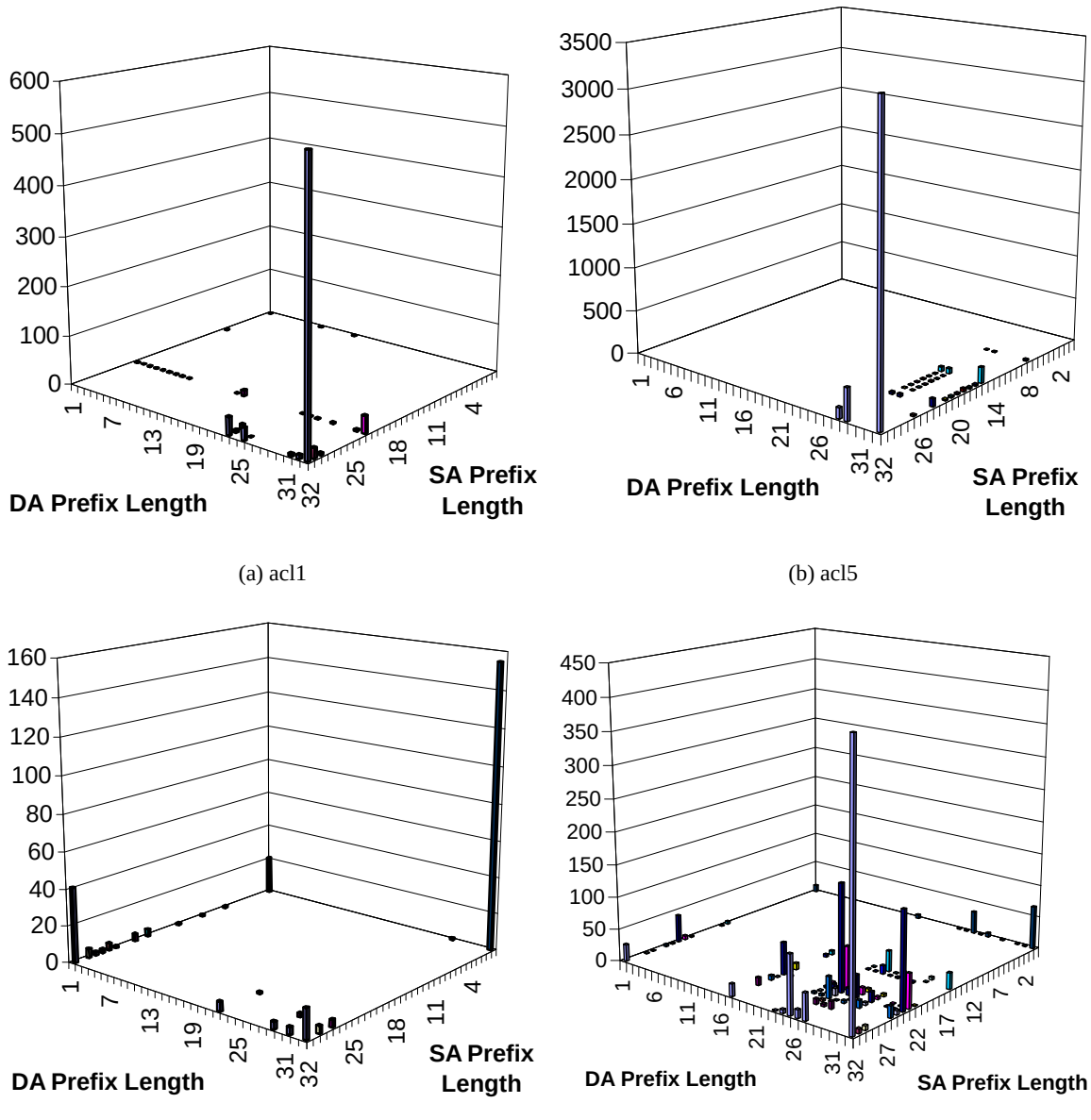
For nodes with two children, we compute *skew*, which is relative measure of the weights of the left and right subtrees of the node. Subtree weight is defined to be the number of filters specifying prefixes in the subtree, not the number of prefixes in the subtree. This definition of weight accounts for "popular" prefixes that occur in many filters. Let $heavy$ be the subtree with the largest weight and let $light$ be the subtree with equal or less weight. The following is a precise definition of skew:

$$skew = 1 - \frac{weight(light)}{weight(heavy)} \tag{1}$$

Note that this definition of skew provides an anonymous measure of address prefix structure, as it does not preserve address prefix values. Consider the following example: given a node k with two children at level m, assume that 10 filters specify prefixes in the 1-subtree of node k (the subtree visited if the next bit of the address is 1) and 25 filters specify prefixes in the 0-subtree of node k. The 1-subtree is the $light$ subtree, the 0-subtree is the $heavy$ subtree, and the skew at node k is 0.6. We compute the average skew for all nodes with two children at level m, record it in the distribution, and move on to level $(m + 1)$. We provide and example of computing skew for the first four levels of an address trie in Figure 5.

The result of this analysis is two distributions for each address trie, a *branching probability* distribution and a *skew* distribution. We plot these distributions for the source address prefixes in filter set acl5 in Figure 6. In Figure 6(a), note that a significant portion of the nodes in levels zero through five have two children, but the amount generally decreases as we move down the trie. The increase at level 16 and 17 is a notable exception. This implies that there is a considerable amount of branching near the "top" of the trie, but the paths generally remain contained as we move down the trie. In Figure 6(b), we observe that skew for nodes with two children hovers around 0.5, thus the one subtree tends to contain prefixes specified by twice as many filters as the other subtree. Note that skew is not defined at levels where all nodes have one child. Also note that levels containing nodes with two children may have an average skew of zero (completely balanced subtrees), but this is rare.

We plot the branching probability and skew for the destination address prefixes specified by filters in filter set acl5 in Figure 7. Note that there is considerably less branching at levels 2 through 11 in the



(a) acl1

(b) acl5

(c) fw1

(d) ipc1

Figure 3: Prefix length distribution for address prefix pairs.

destination address trie; however, there is considerably more branching at lower levels with a majority of the nodes at level 26 having two children. Likewise, the skew is high (when it is defined) at levels 0 through 23, but significantly decreases at levels 24 through 31. Thus destination address prefixes in acl5 will tend to be similar for the first 25 bits or so, then diverge.

Branching probability and skew characterize the structure of the individual source and destination address prefixes; however, it does not capture their interdependence. It is possible that some filters in a filter set match flows contained within a single subnet, while others match flows between different subnets. In order to capture this characteristic of a seed filter set, we measure the "correlation" of source and destination prefixes. In this context, we define correlation to be the probability that the source and destination address

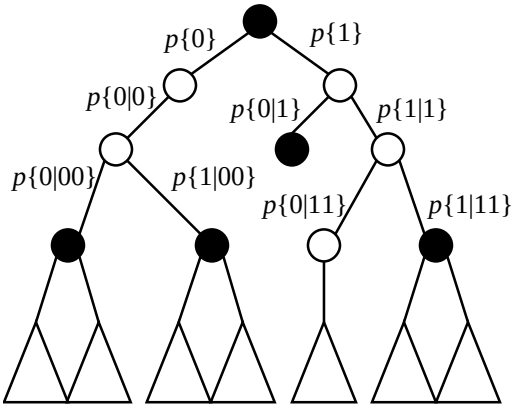


Figure 4: Example of complete statistical characterization of address prefixes.

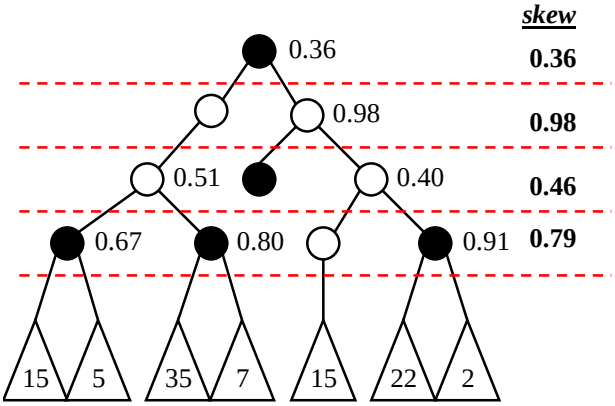
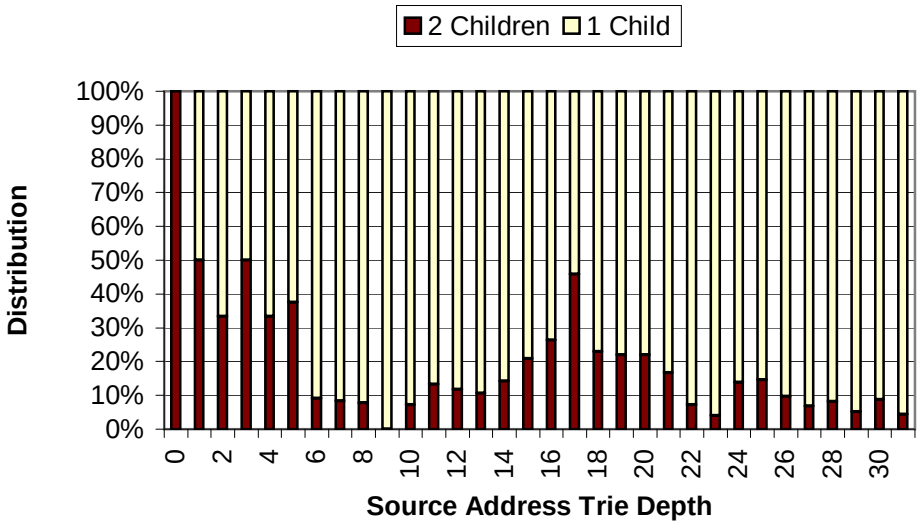


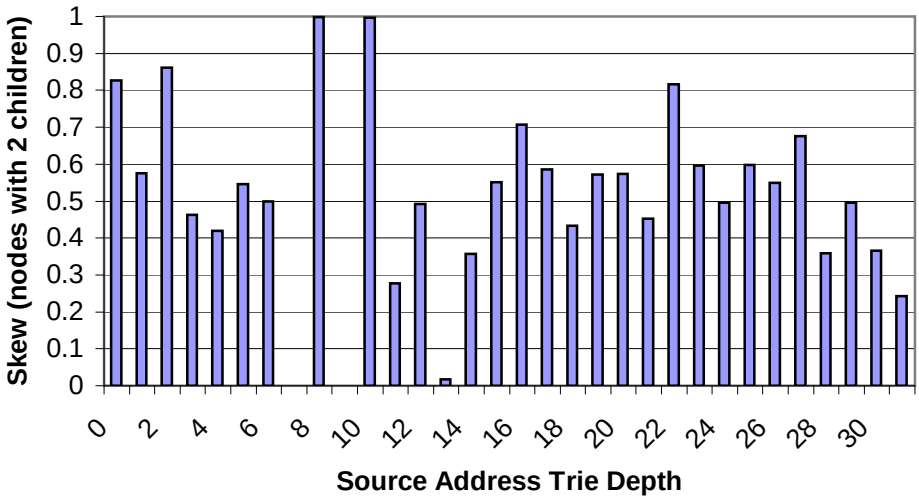
Figure 5: Example of skew computation for the first four levels of an address trie; shaded nodes denote a prefix specified by a single filter; subtrees denoted by triangles with associated weight.

prefixes continue to be the same for a given prefix length. This measure is only valid within the range of address bits specified by both address prefixes. While this measure is not particularly insightful for the purpose of crafting search algorithms, it does allow us to accurately model real filter sets.

Consider the example of a filter set containing filters that all specify flows contained within the same class B network (16-bit network address); the correlation for levels 1 through 16 is 1.0, then falls off for levels 17 through 32 as the source and destination address prefixes diverge. From the seed filter set, we simply generate a probability distribution over the range of possible prefix lengths, $[1 \ldots 32]$. For the filter sets we studied, the address prefix correlation varies widely. The correlation for filter set acl5 is shown in Figure 8(a). Note that approximately 80% of the filters contain source and destination address prefixes with the same first bit. For those with the same first bit, they continue to be identical through the first 13 bits. Of those filters with source and destination address prefixes with the same initial 13 bits, approximately 80% of those continue to be correlated through bit 14, etc. Very few filters in acl5 contain address prefixes the remain correlated through bit 19. The correlation for filter set ipc1 is shown in Figure 8(b). Note that less than half of the filters contain source and destination address prefixes with the same first bit. Likewise, very few filters contain source and destination address prefixes that remain correlated through bit 26.



(a) Source address branching probability; average per level.

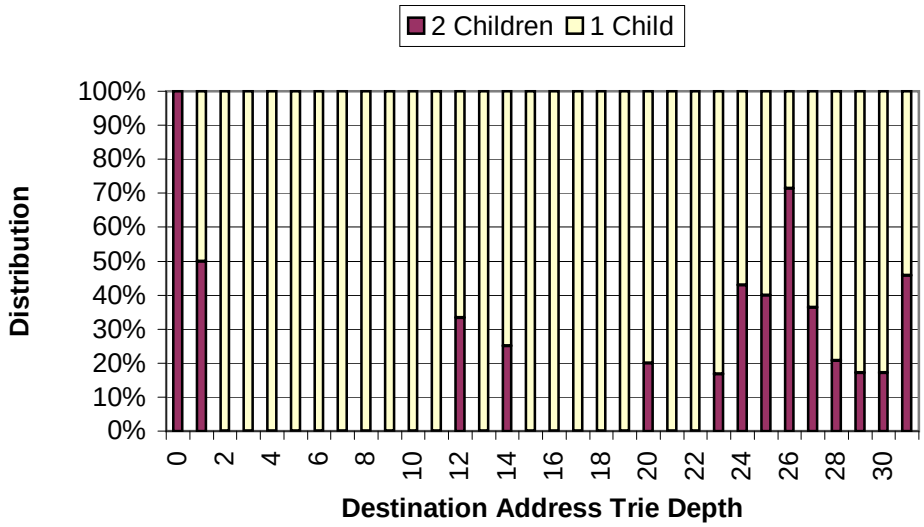


(b) Source address skew; average per level for nodes with two children.

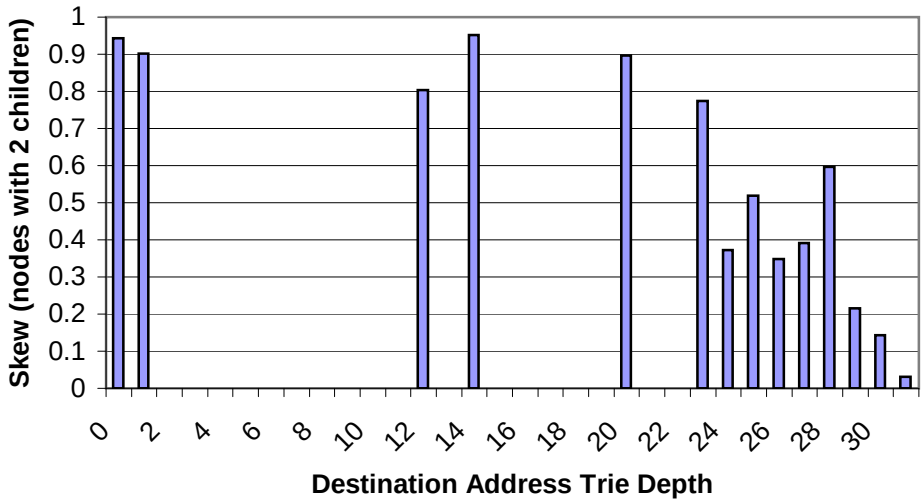
Figure 6: Source address branching probability and skew for filter set acl5.

3.5 Scope

From a geometric perspective, a filter defines a region in d-dimensional space where d is the number of fields specified by the filter. The volume of that region is the product of the one-dimensional "lengths" specified by the filter. For example, length in the source address dimension corresponds to the number of addresses covered by the source address prefix of the filter. Likewise, length in the destination port dimension corresponds to the number of port numbers covered by the destination port range. Points in the d-dimensional space correspond to packet headers; hence, the geometric volume translates to the number of possible packet headers that match the filter. Instead of geometric lengths and volumes, we often refer to these filter properties in terms of a *tuple* specification. To be specific, we define the filter 5-tuple as a vector containing the following fields:



(a) Destination address branching probability; average per level.



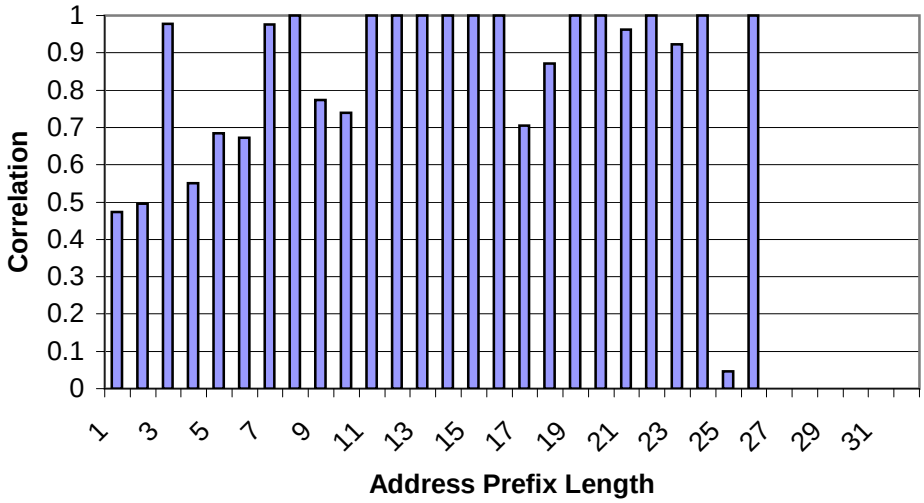
(b) Destination address skew; average per level for nodes with two children.

Figure 7: Destination address branching probability and skew for filter set acl5.

- $t[0]$, source address prefix length, $[0...32]$
- $t[1]$, destination address prefix length, $[0...32]$
- $t[2]$, source port range width, the number of port numbers covered by the range, $[0...2^{16}]$
- $t[3]$, destination port range width, the number of port numbers covered by the range, $[0...2^{16}]$
- $t[4]$, protocol specification, Boolean value denoting whether or not a protocol is specified, $[0, 1]$

The tuple essentially defines the *specificity* of the filter. Filters that are more specific cover a small set of possible packet headers while filters that are less specific cover a large set of possible packet headers. To

(a) acl5.



(b) ipc1

Figure 8: Address prefix correlation; probability that address prefixes of a filter continue to be the same at a given prefix length.

facilitate filter set measurement and modeling, we define a new metric, *scope*, to be logarithmic measure of the number of possible packet headers covered by the filter. Using the 5-tuple definition above, we define *scope* for the 5-tuple as follows:

$$\begin{aligned} scope &= \lg\{(2^{32-t[0]}) \times (2^{32-t[1]}) \times t[2] \times t[3] \times (2^{8(1-t[4])})\} \\ &= (32 - t[0]) + (32 - t[1]) + (\lg t[2]) + (\lg t[2]) + 8(1 - t[4]) \end{aligned} \quad (2)$$

Scope translates the filter tuple into a measure of filter specificity on a scale from 0 to 104. Scope is isomorphic to the logarithm of the geometric volume specified by the filter.

The average 5-tuple scope and standard deviation for the 12 filter sets is shown in Table 4. The average 5-tuple scope ranges from 56 to 24. We note that filters in the ACL filter sets tend to have narrower scope,

Table 4: 5-tuple scope measurements, average (μ_{scope}) and standard deviation (σ_{scope}).

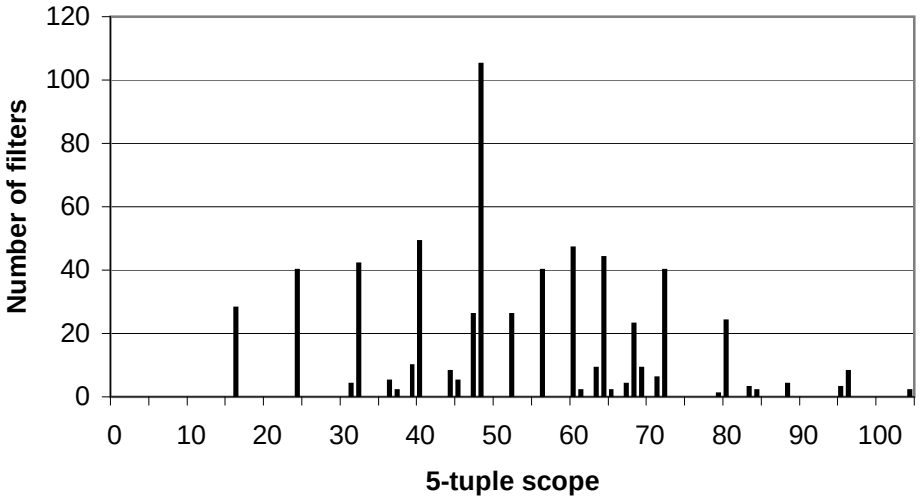
Set	*Size*	μ_{scope}	σ_{scope}
acl1	733	25.0146	13.4585
acl2	623	51.6869	17.6880
acl3	2400	32.0168	15.6699
acl4	3061	30.9481	15.1367
acl5	4557	24.2274	8.0554
fw1	283	51.1686	15.6819
fw2	68	56.5842	23.0965
fw3	184	54.3004	14.8012
fw4	264	48.1127	27.9439
fw5	160	55.7881	16.9506
ipc1	1702	39.7172	19.4508
ipc2	192	47.0521	27.7966

while filters in the FW filter sets tend to have wider scope. While the average scope of the filter sets does not vary drastically, the distributions of filter scope can exhibit drastically different characteristics. Figure 9 shows the 5-tuple scope distribution of filter set acl2 and acl5. The filters in acl2 are distributed among scope values between 16 and 80 with the largest concentration at 48. The filters in acl5 are much more concentrated with most filter distributed among scope values between 16 and 32. The largest concentration is at scope 16.

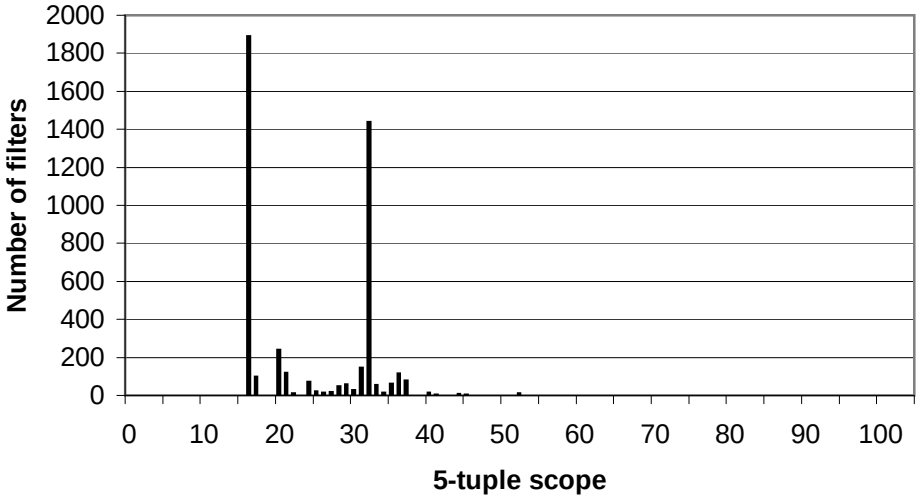
3.6 Additional Fields

An examination of real filter sets reveals that additional fields beyond the standard 5-tuple are relevant. In 10 of the 12 filter sets that we studied, filters contain matches on TCP flags or ICMP type numbers. In most filter sets, a small percentage of the filters specify a non-wildcard value for the flags, typically less then two percent. There are notable exceptions, as approximately half the filters in database *ipc1* contain non-wildcard flags. We argue that new services and administrative policies will demand that packet classification techniques scale to support additional fields (i.e. more "dimensions") beyond the standard 5-tuple. It is not difficult to identify applications that could benefit from packet classification on fields in higher level protocol headers. Consider the following example: an ISP wants to deploy Voice over IP (VoIP) service running over an IPv6/UDP/RTP stack for new IP-enabled handsets and mobile appliances. The ISP also wants to make efficient use of expensive wireless links connecting Base Station Controllers (BSCs) to multiple Base Station Transceivers (BSTs); hence, the ISP would like to use a header compression protocol like Robust Header Compression (ROHC). ROHC is a robust protocol that compresses packet headers for efficient use of wireless links with high loss rates [33]. In order to support this, the BSC must maintain a dynamic filter set which binds packets to ROHC contexts based on fields in the IPv6, UDP, and RTP headers. A total of seven header fields (352 bits) must be examined in order to classify such packets.

Matches on ICMP type number, RTP Synchronization Source Identifier (SSRC), and other higher-level header fields are likely to be exact matches; therefore, the number of unique field values matching any packet are at most two, an exact value and the wildcard if present. There may be other types of matches that more naturally suit the application, such as arbitrary bit masks on TCP flags; however, we do not foresee any reasons why the structure of filters with these additional fields will significantly deviate from the observed structure in current filter tables.



(a) acl2, $\mu = 51.7$, $\sigma = 17.7$



(b) acl5, $\mu = 24.2$, $\sigma = 8.1$

Figure 9: Distribution of 5-tuple scope for filters in filter sets acl2 and acl5.

4 Parameter Files

Given a real filter set, the *Filter Set Analyzer* generates a *parameter file* that contains several statistics and probability distributions that allow the *Filter Set Generator* to produce a synthetic filter set that retains the relevant characteristics of the real filter set. Note that *parameter files* also provide sufficient anonymity of addresses in the original filter set. We chose the statistics and distributions to include in the *parameter file* based on thorough analysis of 12 real filter sets and several iterations of the *Filter Set Generator* design. We have generated a set of 12 *parameter files* which are publicly available along with the *ClassBench* tools suite. There still exists a need for a large sample space of real filter sets from various application environments in order to refine the *parameter files*. By reducing confidentiality concerns, we seek to consolidate efforts to gather filter set samples, enable contributions from a larger body of researchers, and promote the

development of a benchmark set of *parameter files*. It is our hope that *ClassBench* and publicly available *parameter files* eliminates the significant access barrier to realistic test vectors for researchers and designers. We discuss the entries in the parameter file below. Where possible, we avoid discussing format details; interested readers and potential users of *ClassBench* may find a discussion of parameter file format in the documentation provided with the tools.

Protocols The *Filter Set Analyzer* generates a list of the unique protocol specifications and the distribution of filters over those values. We report the protocol distributions from 12 real filter sets and discuss observed trends in Section 3.3.1.

Port Pair Classes As we discussed in Section 3.3.3, we characterize the structure of source and destination port range pairs by defining a *Port Pair Class* (PPC). The *Filter Set Analyzer* generates a PPC distribution for each unique protocol specification in the filter set. This process can be thought of as follows: sort the filters into sets by protocol specification; for each set compute the PPC distribution and record it in the *parameter file*.

Flags For each unique protocol specification in the filter set, the *Filter Set Analyzer* generates a list of unique flag specifications and a distribution of filters over those values. As discussed in Section 3.6, 10 out of the 12 filter sets that we studied contain matches on TCP flags or ICMP type numbers.

Arbitrary Ranges As reported in Section 3.3.2, filter sets typically contain a small number of unique arbitrary range specifications. The *Filter Set Analyzer* generates a list of unique arbitrary range specifications and a distribution of filters over those values for both the source and destination port fields. Both distributions are recorded in the *parameter file*.

Exact Port Numbers As reported in Section 3.3.2, a significant number of filters specify exact port numbers in the source and destination port fields. Like the arbitrary range distributions, the *Filter Set Analyzer* generates a list of unique exact port specifications and a distribution of filters over those values for both the source and destination port fields. Both distributions are recorded in the *parameter file*.

Address Prefix Pair Lengths In Section 3.4 we demonstrated the importance of considering the prefix pair length distribution as opposed to independent distributions for the source and destination address prefix lengths. *Parameter files* represent prefix pair length distributions using a combination of a total prefix length distribution and source prefix length distributions for each specified total length[1] as shown in Figure 10. The total prefix length is simply the sum of the prefix lengths for the source and destination address prefixes. As we will demonstrate in Section 5.2, modeling the total prefix length distribution allows us to easily bias the generation of more or less specific filters based on the *scope* input parameter. The source prefix length distributions associated with each specified total length allow us to model the prefix pair length distribution, as the destination prefix length is simply the difference of the total length and the source length.

Address Prefix Branching and Skew The branching probability and skew distributions defined in Section 3.4 allow us to model the address space coverage and relationships between address prefixes specified

[1]We do not need to store a source prefix distribution for total prefix lengths that are not specified by filters in the filter set.

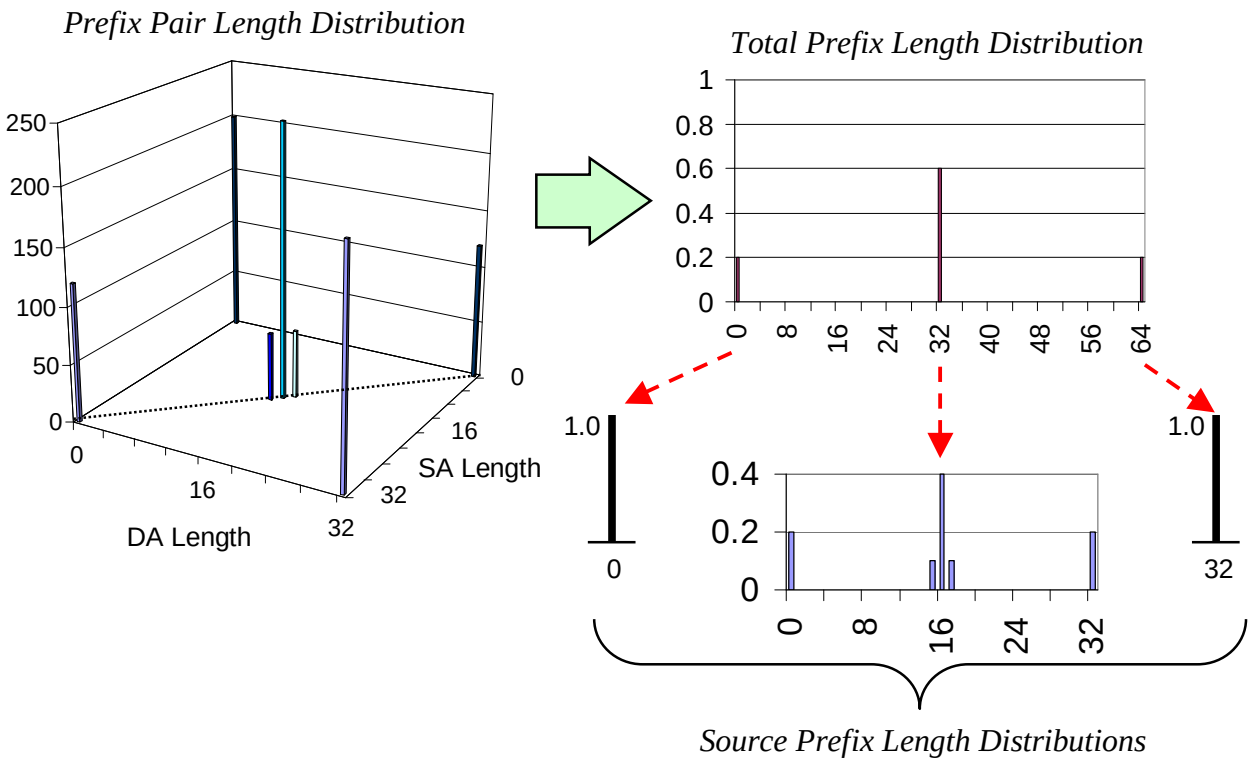


Figure 10: *Parameter files* represent prefix pair length distributions using a combination of a total prefix length distribution and source prefix length distributions for each non-zero total length.

in the filter set. The *Filter Set Analyzer* computes branching probability and skew distributions for both source and destination address prefixes. Both distributions are recorded in the *parameter file*.

Address Prefix Correlation The address correlation distribution defined in Section 3.4 specifies the relationship between source and destination address prefixes in each filter. The *Filter Set Analyzer* computes the address prefix correlation distribution and records it in the *parameter file*.

Prefix Nesting Thresholds The number of unique address prefixes that match a given packet is an important property of real filter sets and is often referred to as *prefix nesting*. We found that if the *Filter Set Generator* is ignorant of this property, it is likely to create filter sets with significantly higher prefix nesting, especially when the synthetic filter set is larger than the filter set used to generate the *parameter file*. Given that prefix nesting remains relatively constant for filter sets of various sizes, we place a limit on the prefix nesting during the filter generation process. The *Filter Set Analyzer* computes the maximum prefix nesting for both the source and destination address prefixes in the filter set and records these statistics in the *parameter file*. The *Filter Set Generator* retains these prefix nesting properties in the synthetic filter set, regardless of size. We discuss the process of generating address prefixes and retaining prefix nesting properties in Section 5.

Scale The *Filter Set Analyzer* also records the size of the real filter set in the generated *parameter file*. This statistic primarily serves as a reference point to users when selecting parameter files to use to test a particular device or algorithm. It is also used when the user chooses to scale the source and destination

address branching probability and skew distributions with filter set size. This option is provided via a high-level command switch to the *Filter Set Generator*. For example, if a parameter file from a firewall filter set of 100 filters is used to generate a synthetic filter set of 10000 filters the user may want to allow the source and destination addresses to cover more of the IP address space while retaining the prefix nesting and prefix pair length distributions.

5 Synthetic Filter Set Generation

The *Filter Set Generator* is the cornerstone of the *ClassBench* tools suite. Perhaps the most succinct way to describe the synthetic filter set generation process is to walk through the pseudocode shown in Figure 11. The first step in the filter generation process is to read the statistics and distributions from the *parameter file*. Rather then list all of the distributions here, we will discuss them when they are used in the process. Next, we get the four high-level input parameters:

- *size*: target size for the synthetic filter set
- *smoothing*: controls the number of new address aggregates (prefix lengths)
- *port scope*: biases the tool to generate more or less specific port range pairs
- *address scope*: biases the tool to generate more or less specific address prefix pairs

We refer to the *size* parameter as a "target" size because the generated filter set may have fewer filters. This is due to the fact that it is possible for the *Filter Set Generator* to produce a filter set containing redundant filters, thus the final step in the process removes the redundant filters. The generation of redundant filters stems from the way the tool assigns source and destination address prefixes that preserve the properties specified in the *parameter file*. This process will be described in more detail in a moment.

Before we begin the generation process, we apply the *smoothing* adjustment to the prefix pair length distributions[2](lines 6 through 10). This adjustment provides a systematic, high-level mechanism for injecting new prefix lengths into the filter set while retaining the general characteristics specified in the *parameter file*. We discuss this adjustment and its effects on the generated filter set in Section 5.1. The *parameter file* specifies a prefix pair length distribution for each Port Pair Class. As described in Section 4, the *parameter file* represents each prefix pair length distribution as a total prefix length distribution with a source prefix length distribution for each specified total length. In order to apply the *smoothing* adjustment, we must iterate over all Port Pair Classes (line 7), apply the adjustment to each total prefix length distribution (line 8) and iterate over all total prefix lengths (line 9), and apply the adjustment to each source prefix length distribution associated with the total prefix length (line 10).

Prior to generating filters, we allocate a temporary array (line 11). The next set of steps (lines 12 through 27) generate a *partial* filter for each entry in the `Filters` array. Basically, we assign all filter fields except the address prefix values. Note that the prefix lengths for both source and destination address *are* assigned. The reason for this approach will become clear when we discuss the assignment of address prefix values in a moment. The first step in generating a *partial* filter is to select a protocol from the `Protocols` distribution specified by the *parameter file* (line 14). Note that this selection is performed with a uniform random variable, `rv` (line 13). We chose to select the protocol first because we found that the protocol specification dictates the structure of the other filter fields. Next, we select the protocol flags from the `Flags` distribution

[2]Note that the *scope* adjustments do not add any new prefix lengths to the distributions. It only changes the likelihood that longer or shorter prefix lengths in the distribution are chosen.

```
FilterSetGenerator()
   // Read input file and parameters
1  read(parameter file)
2  get(size)
3  get(smoothing)
4  get(port scope)
5  get(address scope)
   // Apply smoothing to prefix pair length distributions
6  If smoothing > 0
7     For i : 1 to MaxPortPairClass
8        TotalLengths[i]→smooth(smoothing)
9        For j : 0 to 64
10          SALengths[i][j]→smooth(smoothing)
   // Allocate temporary filter array
11 FilterType Filters[size]
   // Generate filters
12 For i : 1 to size
       // Choose an application specification
13     rv = Random()
14     Filters[i].Prot = Protocols→choose(rv)
15     rv = Random()
16     Filters[i].Flags = Flags[Filters[i].Prot]→choose(rv)
17     rv = RandomBias(port scope)
18     PPC = PPCMatrix[Filters[i].Prot]→choose(rv)
19     rv = Random()
20     Filters[i].SP = SrcPorts[PPC.SPClass]→choose(rv)
21     rv = Random()
22     Filters[i].DP = DstPorts[PPC.DPClass]→choose(rv)
       // Choose an address prefix length pair
23     rv = RandomBias(address scope)
24     TotalLength = TotalLengths[PPC]→choose(rv)
25     rv = Random()
26     Filters[i].SALength = SrcLengths[PPC][TotalLength]→choose(rv)
27     Filters[i].DALength = TotalLength - Filters[i].SALength
   // Assign address prefix pairs
28 AssignSA(Filters)
29 AssignDA(Filters)
   // Remove redundant filters and prevent filter nesting
30 RemoveRedundantFilters(Filters)
31 OrderNestedFilters(Filters)
32 PrintFilters(Filters)
```

Figure 11: Pseudocode for *Filter Set Generator*.

associated with the chosen protocol (line 16). The `Flags` distributions for all protocol specifications are given by the *parameter file*. Note that the protocol flags field is typically the wildcard unless the chosen

protocol is TCP or ICMP. This selection is also performed with a uniform random variable (line 15).

After choosing the protocol and flags, we select a Port Pair Class, `PPC`, from the Port Pair Class matrix, `PPCMatrix`, associated with the chosen protocol (line 18). As discussed in Section 3.3.3, Port Pair Classes specify the type of port range specified by the source and destination port fields (wildcard, arbitrary range, etc.). Note that the selection of the `PPC` is performed with a random variable that is biased by the *port scope* parameter (line 17). This adjustment allows the user to bias the *Filter Set Generator* to produce a filter set with more or less specific Port Pair Classes where WC-WC (both port ranges wildcarded) is the least specific and EM-EM (both port ranges specify an exact match port number) is the most specific. We discuss this adjustment and its effects on the generated filter set in Section 5.2. Given the Port Pair Class, we can select the source and destination port ranges from their respective port range distributions associated with each Port Class (lines 20 and 22). Note that the distributions for Port Classes WC, HI, and LO are trivial as they define single ranges; therefore, the *parameter file* only needs to specify arbitrary range (AR) and exact match (EM) port number distributions for both the source and destination ports. The selection of port ranges from a Port Class distribution is performed using a uniform random variable (lines 19 and 21).

Selecting the address prefix pair lengths is the last step in generating a *partial* filter. We select a total prefix pair length from the distribution associated with the chosen Port Pair Class (line 24) using a random variable biased by the *address scope* parameter (line 23). We discuss this adjustment and its effects on the generated filter set in Section 5.2. We select a source prefix length from the distribution associated with the chosen Port Pair Class and total length (line 26) using a uniform random variable (line 25). Note that we use an unbiased, uniform random variable for choosing the source address length. This allows us to retain the relationships between source and destination address prefix lengths. Finally, we calculate the destination address prefix length using the chosen total length and source address prefix length (line 27).

After we generate all the *partial* filters, we must assign the source and destination address prefix values. We begin by assigning the source address prefix values (line 28). The `AssignSA` routine constructs a binary trie using the set of source address prefix lengths in `Filters` and the source address branching probability and skew distributions specified by the *parameter file*. We start by allocating a root node, constructing a list of filters `FilterList` containing all the partial filters in `Filters`, and passing `FilterList` and a node pointer to a recursive process, `VisitNode`. This process first examines all of the entries in `FilterList`. If an entry has a source prefix length equal to the level of the node[3], it assigns the node's address to the entry and removes the entry from `FilterList`. Once completed, `VisitNode` recursively distributes the remaining filters to child nodes according to the branching probability and skew for the node's level. Note that we also keep track of the number of prefixes that have been assigned along a path by passing a `Nest` variable to the recursive process. If `Nest` $\geq$ `SANestThresh - 1`, where `SANestThresh` is the source prefix nesting threshold specified by the *parameter file*, then `VisitNode` ignores the branching probability and skew distributions. In this case, `VisitNode` partitions `FilterList` into two lists, one containing filters with source address prefix lengths equal to the next tree level, and one containing all the remaining filters. `VisitNode` then recursively passes the lists to two child nodes. In doing so, we ensure that the nesting threshold is not exceeded.

Assigning destination address prefix values is symmetric to the process for source address prefixes with one extension. In order to preserve the relationship between source and destination address prefixes in each filter, the `AssignDA` process (line 29) also considers the correlation distribution specified in the *parameter file*. In order to preserve the correlation, `AssignDA` employs a two-phase process of constructing the destination address trie. The first phase recursively distributes filters according to the correlation distribution. When the address prefixes of a particular filter cease to be correlated, it stores the filter in a temporary `StubList` associated with the current tree node. The second phase recursively walks down the tree and

[3]Node level is synonymous with tree depth.

completes the assignment process in the same manner as the `AssignSA` process, with the exception that the `StubList` is appended to the `FilterList` passed to the `AssignDA` process prior to processing.

Note that we do not explicitly prevent the *Filter Set Generator* from generating redundant filters. Identical *partial* filters may be assigned the same source and destination address prefix values by the `AssignSA` and `AssignDA` functions. In essence, this preserves the characteristics specified by the *parameter file* because the number of unique filter field values allowed by the various distributions is inherently limited. Consider the example of attempting to generate a large filter set using a *parameter file* from a small filter set. If we are forced to generate the number of filters specified by the *size* parameter, we face two unfavorable results: (1) the resulting filter set may not model the *parameter file* because we are repeatedly forced to choose values from the tails of the distributions in order to create unique filters, or (2) the *Filter Set Generator* never terminates because it has exhausted the distributions and cannot create any more unique filters. With the current design of the *Filter Set Generator*, a user can produce a larger filter set by simply increasing the *size* target beyond the desired size. While this does introduce some variability in the size of the synthetic filter set, we believe this is a tolerable trade-off to make for maintaining the characteristics in the *parameter file* and achieving reasonable execution times for the *Filter Set Generator*.

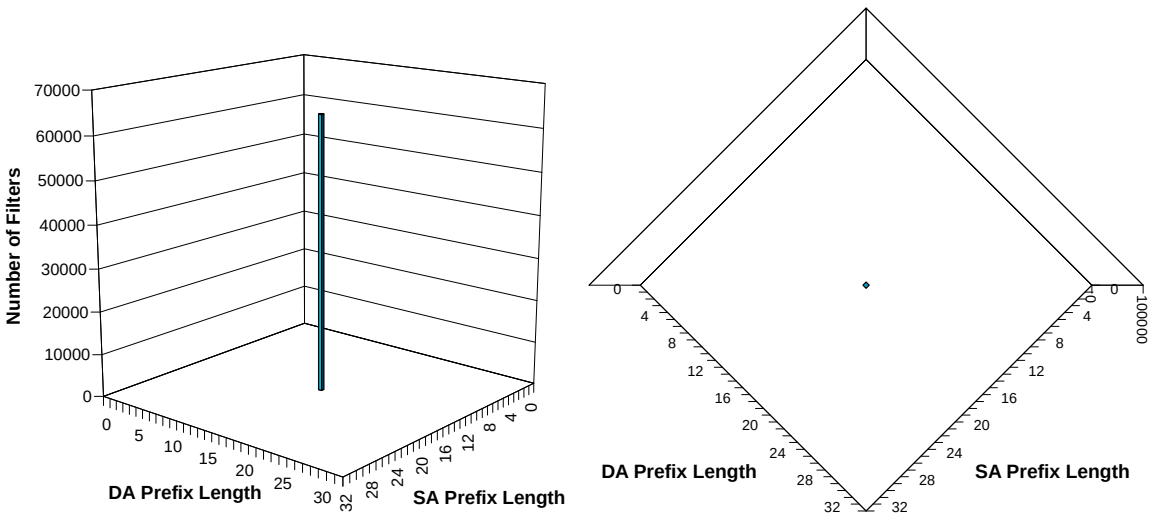
Thus, after generating a list of *size* synthetic filters, we remove any redundant filters from the list via the `RemoveRedundantFilters` function (line 30). A naïve implementation of this function would require $O(N^2)$ time, where N is equal to $size$. We discuss an efficient mechanism for removing redundant filters from the set in Section 5.3. After removing redundant filters from the filter set, we sort the filters in order of increasing scope (line 31). This allows the filter set to be searched using a simple linear search technique, as nested filters will be searched in order of decreasing specificity. An efficient technique for performing this sorting step is also discussed in Section 5.3. Finally, we print the filter set to an output file (line 32). The following subsections provide detailed descriptions and analyses of the smoothing and scope adjustments, as well as efficient techniques for removing redundant filters and sorting the filters to prevent nesting.

5.1 Smoothing Adjustment

As filter sets scale in size, we anticipate that new address prefix pair lengths will emerge due to network address aggregation and segregation. In order to model this behavior, we provide for the introduction of new prefix lengths in a structured manner. Injecting purely random address prefix pair lengths in the distributions during the generation process neglects the structure of the filter set used to generate the *parameter file*. Using scope as a measure of distance, we expect that new address aggregates will emerge "near" an existing address aggregate. Consider the address prefix pair length distribution shown in Figure 12. In this example, all filters in the filter set have 16-bit source and destination address prefixes; thus, the distribution is a single "spike". When injecting new address prefix pair lengths into the distribution, we would like them to be clustered around the existing spike in the distribution. This structured approach translates "spikes" in the distribution into smoother "hills"; hence, we refer to the process as smoothing.

In order to control the injection of new prefix lengths, we define a *smoothing* parameter which limits the maximum radius of deviation from the original prefix pair length, where radius is measured in the number of bits specified by the prefix pair. Geometrically, this measurement may be viewed as the Manhattan distance from one prefix pair length to another. For convenience, let the *smoothing* parameter be equal to r. We chose to model the clustering using a symmetric binomial distribution. Given the parameter r, a symmetric binomial distribution is defined on the range $[0 : 2r]$, and the probability at each point i in the range is given by:

$$p_i = \begin{pmatrix} 2r \\ i \end{pmatrix} \left(\frac{1}{2}\right)^{2r} \tag{3}$$



(a) $r = 0$ (b) $r = 0$, top view

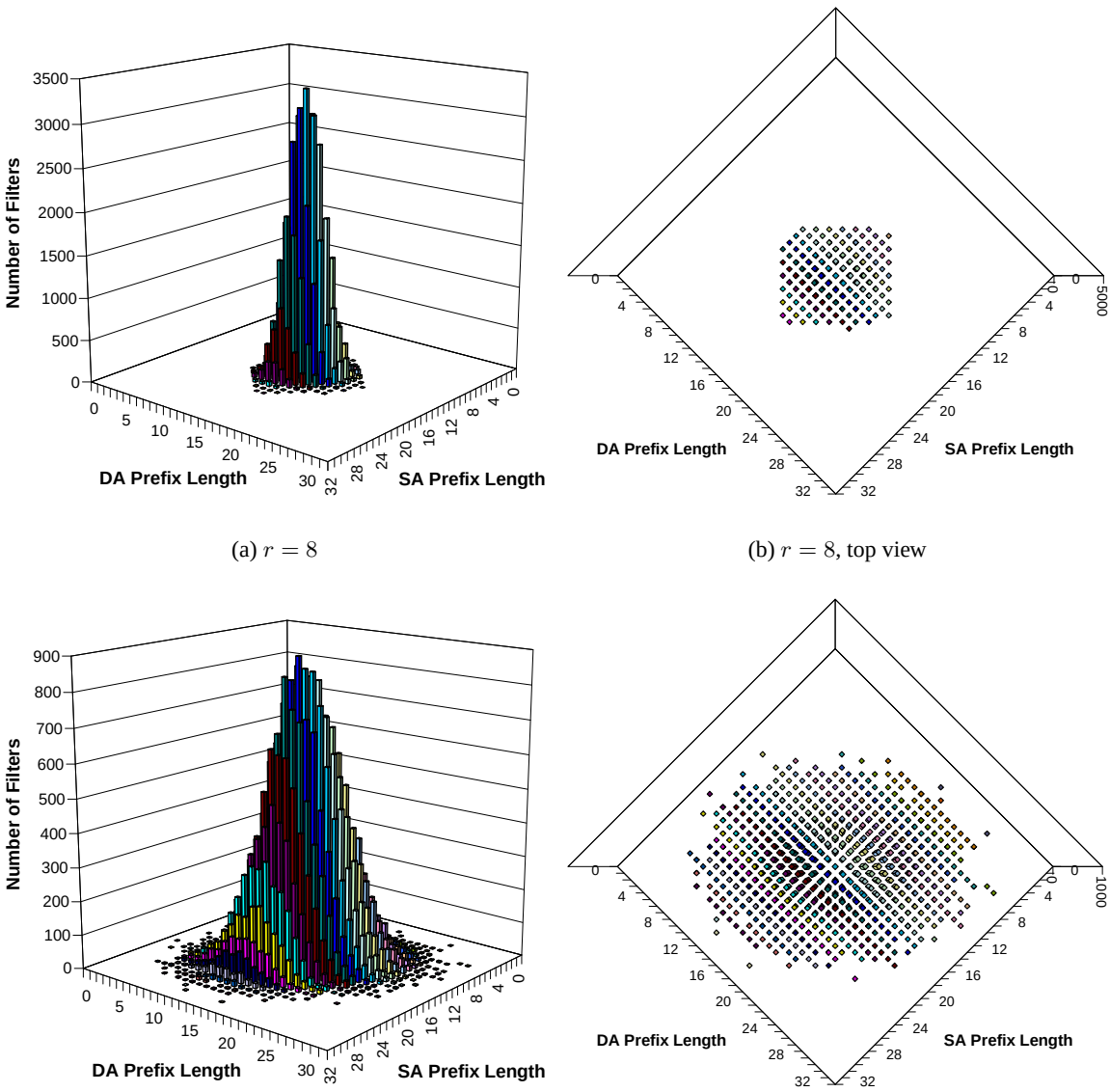
Figure 12: Prefix pair length distribution for a synthetic filter set of 64000 filters generated with a *parameter file* specifying 16-bit prefix lengths for all addresses.

Note that r is the median point in the range with probability p_r, and r may assume values in the range $[0 : 64]$.

Once we generate the symmetric binomial distribution from the *smoothing* parameter, we apply this distribution to each specified prefix pair length. The smoothing process involves scaling each "spike" in the distribution according to the median probability p_r, and binomially distributing the residue to the prefix pair lengths within the r-bit radius. When prefix lengths are at the "edges" of the distribution, we simply truncate the binomial distribution. This requires us to normalize the prefix pair length distribution as the last step in the smoothing process. Note that we must apply the smoothing adjustment to each prefix pair length distribution associated with each Port Pair Class in the *parameter file*. In order to demonstrate this process, we provide an example of smoothing the prefix pair length distribution in Figure 12 using two different values of r. Figure 13(a) and Figure 13(b) show the prefix pair length distributions for a synthetic filter set generated with a *parameter file* specifying 16-bit prefix lengths for all addresses and a smoothing parameter $r = 8$. With the exception of the fringe effects due to random number generation, the single spike at 16-16 is binomially distributed to the prefix pair lengths within a Manhattan distance of 8. The same effect is shown in Figure 13(a) and Figure 13(b) for a smoothing parameter $r = 32$.

In practice, we expect that the *smoothing* parameter will be limited to at most 8. In order to demonstrate the effect of smoothing in a realistic context, we generated a synthetic filter set using a *smoothing* parameter of 4. Figure 14(a) and Figure 14(b) show the prefix pair length distribution for a synthetic filter set of 64000 filters generated using the ipc1 *parameter file* and smoothing parameter $r = 0$. Figure 14(c) and Figure 14(d) show the prefix pair length distribution for a synthetic filter set of 64000 filters generated using the ipc1 *parameter file* and smoothing parameter $r = 4$. Note that this synthetic filter set retains the structure of the original filter set while modeling a realistic amount of address aggregation and segregation.

Recall that we choose to truncate and normalize to deal with the edge cases. As evident in Figure 14, many of the most common address prefix pair lengths occur at the edges of the distribution. As a result, applying the smoothing adjustment may affect the average scope of the generated filter set. Consider the case of the spike at 32-32 (fully specified source and destination addresses). Applying the smoothing adjustment



(a) $r = 8$

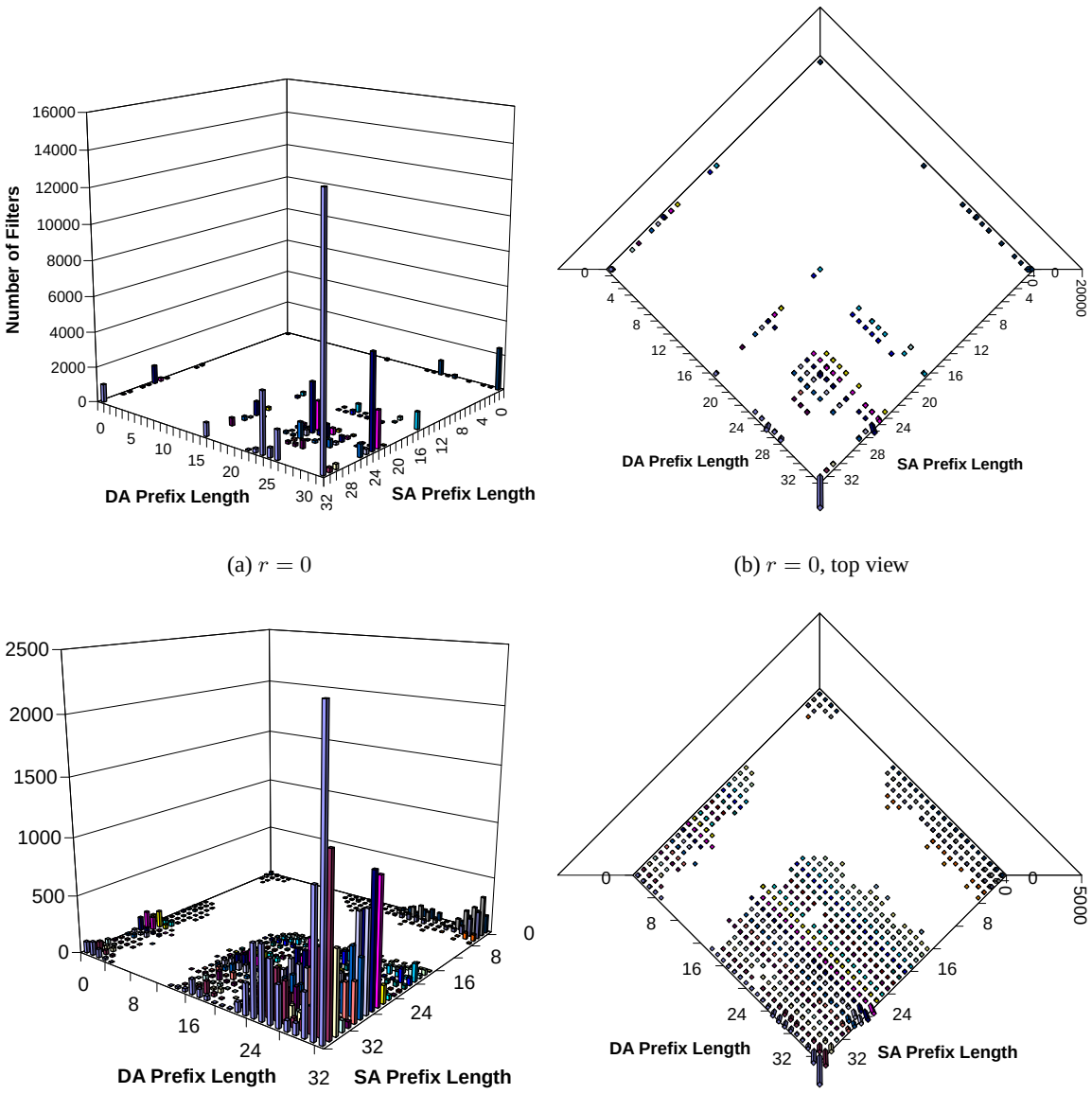
(b) $r = 8$, top view

(c) $r = 32$

(d) $r = 32$, top view

Figure 13: Prefix pair length distributions for a synthetic filter set of 64000 filters generated with a *parameter file* specifying 16-bit prefix lengths for all addresses and various values of smoothing parameter r.

to this point distributes some of the residue to less specific prefix pair lengths, but the residue allocated to more specific prefix pair lengths is truncated as there are not any more specific prefix pair lengths. In order to assess the effects of truncation and normalization on the resulting filter sets, we generated several filter sets of the same size using three different *parameter files* and various values of the smoothing parameter. The results are shown in Figure 5.1. Note that as we increase the amount of smoothing applied to the prefix pair length distributions, the effect on the 5-tuple scope and address pair scope is minimal. We observe a slight drift toward the median scope value due to the aforementioned truncation of the distributions at the edges.



(a) $r = 0$

(b) $r = 0$, top view

(c) $r = 4$

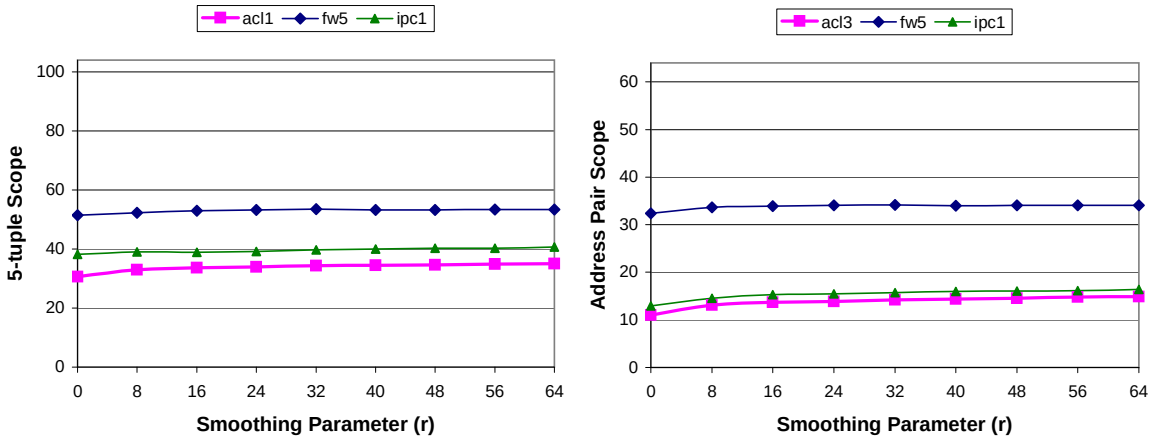
(d) $r = 4$, top view

Figure 14: Prefix pair length distribution for a synthetic filter set of 64000 filters generated with the ipc1 *parameter file* with smoothing parameters $r = 0$ and $r = 4$.

5.2 Scope Adjustment

As filter sets scale in size and new applications emerge, it is likely that the average scope of the filter set will change. As the number of flow-specific filters in a filter sets increases, the specificity of the filter set increases and the average scope decreases. If the number of explicitly blocked ports for all packets in a firewall filter set increases, then the specificity of the filter set may decrease and the average scope may increase[4]. In order to explore the effect of filter scope on the performance of algorithms and packet classification devices,

[4]We are assuming a common practice of specifying an exact match on the blocked port number and wildcards for all other filter fields



(a) 5-tuple Scope

(b) Address Prefix Pair Scope

Figure 15: Average scope of synthetic filter sets consisting of 16000 filters generated with parameter files extracted from filter sets *acl3*, *fw5*, and *ipc1*, and various values of the smoothing parameter r.

we provide high-level adjustments of the average scope of the synthetic filter set. Two input parameters, *address scope* and *port scope*, allow the user to bias the *Filter Set Generator* to create more or less specific address prefix pairs and port pairs, respectively.

In order to illustrate the effects of scope adjustments, consider the standard method of sampling from a distribution using a uniformly distributed random variable. In Figure 16, we show the cumulative distribution for the total prefix pair length associated with the WC-WC port pair class of the acl2 filter set. In order to sample from this distribution, the *Filter Set Generator* selects a random number between zero and one using a uniform random number generator, then chooses the total prefix pair length covering that number in the cumulative distribution. Graphically, this amounts to projecting a horizontal line from the random number on the y-axis. The "step" which it intersects is the sampled total prefix pair length. In Figure 16, we shown an example of sampling with a random variable equal to 0.5 to choose the total prefix pair length of 44.

The *address scope* adjustment essentially biases the sampling process to select more or less specific total prefix pair lengths. We can realize this in two ways: (1) apply the adjustment to the cumulative distribution, or (2) bias the random variable used to sample from the cumulative distribution. The first option requires that we recompute the cumulative density distribution to make longer or shorter prefix lengths more or less probable, as dictated by the *address scope* parameter. The second option provides a conceptually simpler alternative. Returning to the example in Figure 16, if we want to bias the *Filter Set Generator* to produce more specific address prefix pairs, then we want the random variable used to sample from the distribution to be biased to values closer to 1. The reverse is true if we want less specific address prefix pairs. Thus, in order to apply the scope adjustment we simply use a random number generator to choose a uniformly distributed random variable, rv_{uni}, apply a biasing function to generate a biased random variable, rv_{bias}, and sample from the cumulative distribution using rv_{bias}.

While there are many possible biasing functions, we limit ourselves to a particularly simple class of functions. Our chosen biasing function may be viewed as applying a slope, s, to the uniform distribution as shown in Figure 17(a). When the slope $s = 0$, the distribution is uniform. The biased random variable corresponding to a uniform random variable on the x-axis is equal to the area of the rectangle defined by the value and a line intersecting the y-axis at one with a slope of zero. Thus, the biased random variable is equal

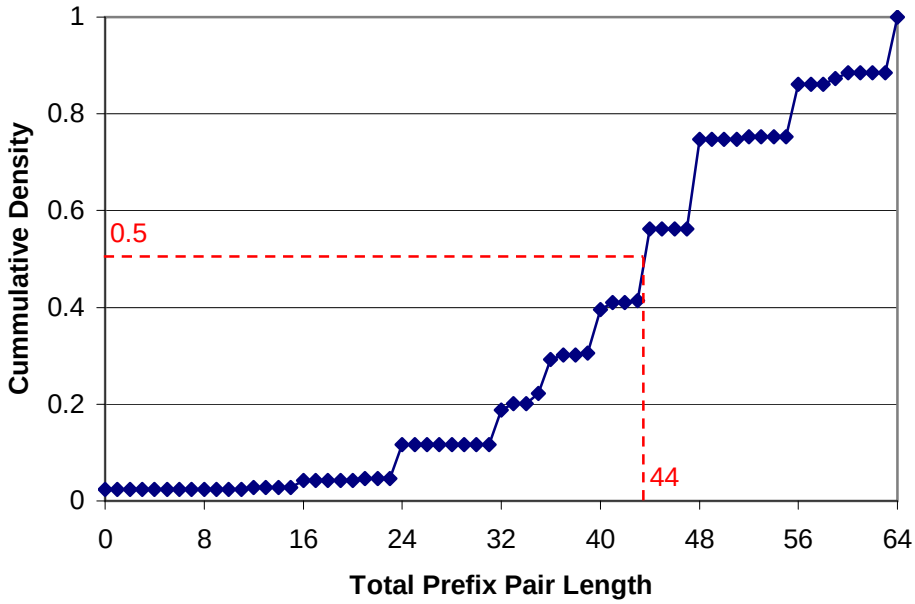


Figure 16: Example of sampling from a cumulative distribution using a random variable. Distribution is for the total prefix pair length associated with the WC-WC port pair class of the acl2 filter set. A random variable equal to 0.5 chooses 44 as the total prefix pair length.

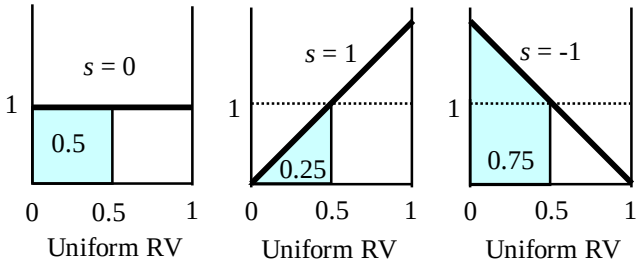
to the uniform random variable. As shown in Figure 17(a), we can bias the random variable by altering the slope of the line. Note that in order for the biasing function to be defined for random variables in the range $[0:1]$ and have a cumulative probability of 1 for a random variable equal to 1, the slope adjustment must be in the range $[-2:2]$. Graphically, this results in the line pivoting about the point $(0.5, 1)$. For convenience, we define the scope adjustments to be in the range $[-1:1]$, thus the slope is equal to two times the scope adjustment. For non-zero slope values, the biased random variable corresponding to a uniform random variable on the x-axis is equal to the area of the trapezoid[5] defined by the value and a line intersecting the point $(0.5, 1)$ with a slope of s. The expression for the biased random variable, rv_{bias}, given a uniform random variable, rv_{uni}, and a *scope* parameter in the range $[-1:1]$ is:

$$rv_{bias} = rv_{uni}(scope \times rv_{uni} - scope + 1) \tag{4}$$

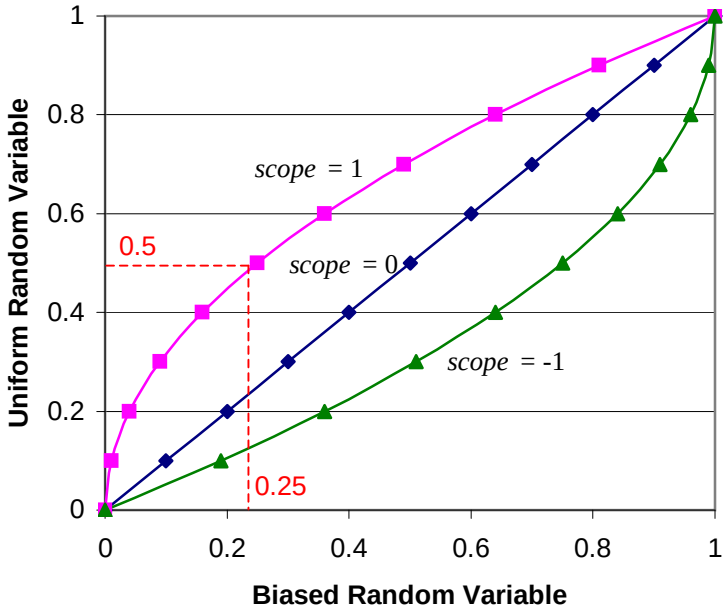
Figure 17(b) shows a plot of the biasing function for *scope* values of 0, -1, and 1. We also provide a graphical example of computing the biased random variable given a uniform random variable of 0.5 and a *scope* parameter of 1. In this case the rv_{bias} is 0.25. Let us return to the example of choosing the total address prefix length from the cumulative distribution. In Figure 18, we show examples of sampling the distribution using the unbiased uniform random variable, $rv_{uni} = 0.5$, and the biased random variable, $rv_{bias} = 0.25$, resulting from applying the biasing function with $scope = 1$. Note that the biasing results in the selection of a less specific address prefix pair, a total length of 35 as opposed to 44.

Positive values of *address scope* bias the *Filter Set Generator* to choose less specific address prefix pairs, thus increasing the average scope of the filter set. Likewise, negative values of *address scope* bias the *Filter Set Generator* to choose more specific address prefix pairs, thus decreasing the average scope of the filter set. The same effects are realized by the *port scope* adjustment by biasing the *Filter Set Generator* to select

[5]Recall that the area of a trapezoid is one half the product of the height and the sum of the lengths of the parallel edges, $A = \frac{1}{2} \times h \times (l_1 + l_2)$.



(a) Biased random variable is defined by area under line with slope $s = 2 \times scope$.



(b) Plot of scope biasing function.

Figure 17: Scope applies a biasing function to a uniform random variable.

more or less specific port range pairs. Note that the cumulative distribution must be constructed in such a way that the distribution is computed over values sorted from least specific to most specific.

Finally, we report the results of tests assessing the effects of the *address scope* and *port scope* parameters on the synthetic filter sets generated by the *Filter Set Generator*. Each data point in the plots in Figure 5.2 is from a synthetic filter set containing 16000 filters generated from a *parameter file* from filter sets acl3, fw5, or ipc1. Figure 19(a) shows the effect of the *address scope* parameter on the average scope of the address prefix pairs in the resulting filter set. Over its range of values, the *address scope* alters the average address pair scope by ± 4 to ± 6. Figure 19(b) shows the effect of the *port scope* parameter on the average scope of the port range pairs in the resulting filter set. Over its range of values, the *port scope* alters the average port pair scope by ± 1.5 to ± 2.5. Note that the magnitude of change in average scope for both parameters is approximately the same relative to the range of possible scope values. Figure 19(c) shows the effect of both scope parameters on the average scope of the filters in the resulting filter set. For these tests, both scope parameters were set to the same value. Over their range of values, the scope parameters alter

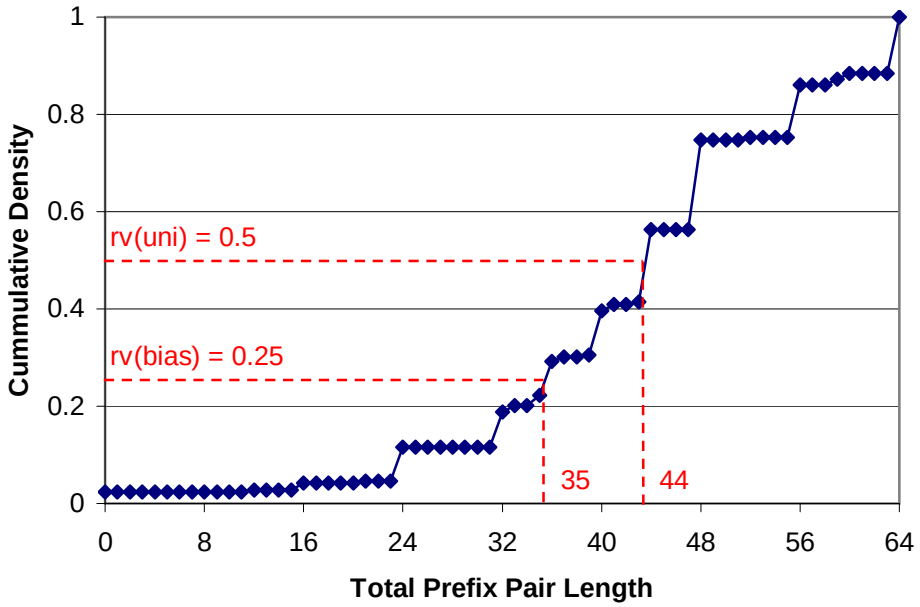


Figure 18: Example of sampling from a cumulative distribution using a random variable. Distribution is for the total prefix pair length associated with the WC-WC port pair class of the acl2 filter set. A random variable equal to 0.5 chooses 44 as the total prefix pair length.

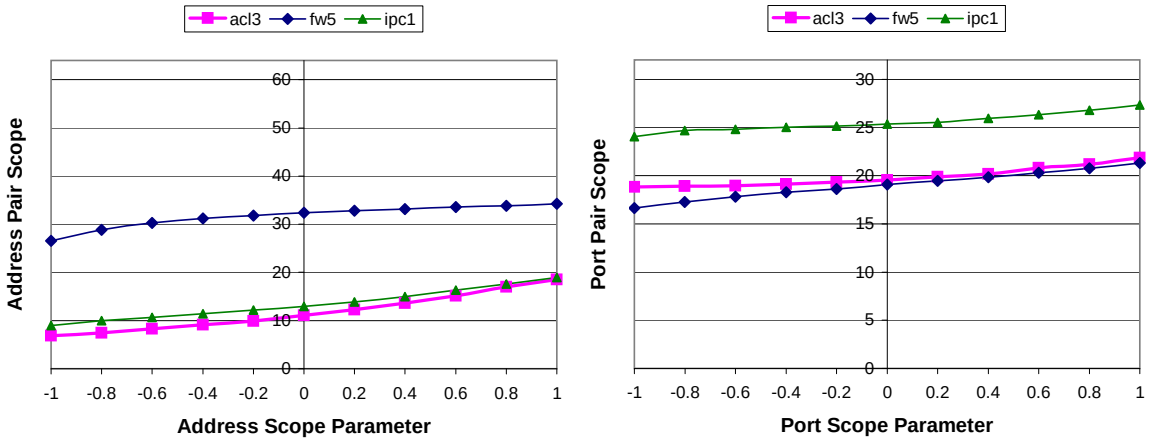
the average filter scope by ± 6 to ± 7.5. We assert that these scope adjustments provide a convenient high-level mechanism for exploring the effects of filter specificity on the performance of packet classification algorithms and devices.

5.3 Filter Redundancy & Priority

The final steps in synthetic filter set generation are removing redundant filters and ordering the remaining filters in order of increasing scope. The removal of redundant filters may be realized by simply comparing each filter against all other filters in the set; however, this naïve implementation requires $O(N^2)$ time, where N is equal to $size$. Such an approach makes execution times of the *Filter Set Generator* prohibitively long for filter sets in excess of a few thousand filters. In order to accelerate this process, we first sort the filters into sets according to their tuple specification. Sorting filters into tuple sets was introduced by Srinivasan, et. al. in the context of the *Tuple Space Search* packet classification algorithm [34].

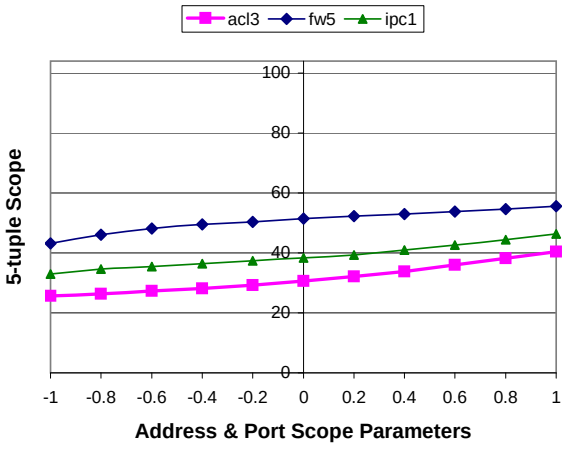
We perform this sorting efficiently by constructing a binary search tree of tuple set pointers, using the scope of the tuple as the key for the node. When adding a filter to a tuple set, we search the set for redundant filters. If no redundant filters exist in the set, then we add the filter to the set. If a redundant filter exists in the set, we discard the filter. The time complexity of this search technique depends on the number of tuples created by filters in the filter set and the distribution of filters across the tuples. In practice, we find that this technique provides acceptable performance. Generating a synthetic filter set of 10k filters requires approximately five seconds, while a filter set of 100k filters requires approximately five minutes with a Sun Ultra 10 workstation.

In order to support the traditional linear search technique, filter priority is often inferred by placement in an ordered list. In such cases, the first matching filter is the best matching filter. This arrangement could obviate a filter f_i if a less specific filter $f_j \supset f_i$ occupies a higher position in the list. To prevent this, we



(a) Effect of *address scope* adjustment on the address prefix pair scope

(b) Effect of *port scope* adjustment on the port pair scope



(c) 5-d Scope

Figure 19: Average scope of synthetic filter sets consisting of 16000 filters generated with parameter files extracted from filter sets *acl3*, *fw5*, and *ipc1*, and various values of the scope parameters.

order the filters in the synthetic filter set according to scope, where filters with minimum scope occur first. The binary search tree of tuple set pointers makes this ordering task simple. Recall that we use scope as the node key. Thus, we simply perform an in-order walk of the binary search tree, appending the filters in each tuple set to the output list of filters.

6 Trace Generation

When benchmarking a particular packet classification algorithm or device, many of the metrics of interest such as storage efficiency and maximum decision tree depth may be garnered using the synthetic filter sets generated by the *Filter Set Generator*. In order to evaluate the throughput of techniques employing caching or the power consumption of various devices under load, we must exercise the algorithm or device using a

```
TraceGenerator()
   // Generate list of synthetic packet headers
1  read(FilterSet)
2  get(scale)
3  get(ParetoA)
4  get(ParetoB)
5  Threshold = scale × size(FilterSet)
6  HeaderList Headers()
7  While size(Headers) < Threshold
8        RandFilt = randint(0,size(FilterSet))
9        NewHeader = RandomCorner(RandFilt,FilterSet)
10       Copies = Pareto(ParetoA,ParetoB)
11       For i : 1 to Copies
12          Headers→append(NewHeader)
13 Headers→print
```

Figure 20: Pseudocode for *Trace Generator*.

sequence of synthetic packet headers. The *Trace Generator* produces a list of synthetic packet headers that probe filters in a given filter set. Note that we do not want to generate random packet headers. Rather, we want to ensure that a packet header is covered by at least one filter in the *FilterSet* in order to exercise the packet classifier and avoid default filter matches. We experimented with a number of techniques to generate synthetic headers. One possibility is to compute all the d-dimensional polyhedra defined by the intersections of the filters in the filter set, then choose a point in the d-dimensional space covered by the polyhedra. The point defines a packet header. The best-matching filter for the packet header is simply the highest priority filter associated with the polyhedra. If we generate at least one header corresponding to each polyhedra, we fully exercise the filter set. The number of polyhedra defined by filter intersections grows exponentially, and thus fully exercising the filter set quickly becomes intractable. As a result, we chose a method that partially exercises the filter set and allows the user to vary the size and composition of the headers in the trace using high-level input parameters. These parameters control the scale of the header trace relative to the filter set, as well as the locality of reference in the sequence of headers. As we did with the *Filter Set Generator*, we discuss the *Trace Generator* using the pseudocode shown in Figure 20.

We begin by reading the *FilterSet* from an input file (line 1). Next, we get the input parameters *scale*, *ParetoA*, and *ParetoB* (lines 2 through 4). The *scale* parameter is used to set a threshold for the size of the list of headers relative to the size of the *FilterSet* (line 5). In this context, *scale* specifies the ratio of the number of headers in the trace to the number of filters in the filter set. After computing the `Threshold`, we allocate a list of headers, `Headers` (line 6). The next set of steps continue to generate synthetic headers as long as the size of `Headers` does not exceed the `Threshold`.

Each iteration of the header generation loop begins by selecting a random filter in the *FilterSet* (line 8). Next, we must choose a packet header covered by the filter. In the interest of exercising priority resolution mechanisms and providing conservative performance estimates for algorithms relying on filter overlap properties, we would like to choose headers matching a large number of filters. In the course of our analyses, we found the number of overlapping filters is large for packet headers representing the "corners" of filters. When we view a filter as defining a d-dimensional rectangle, the corners of this rectangle represent points in the d-dimensional space which correspond to packet headers. Each field of a filter covers a range of values.

Choosing a packet header corresponding to a "corner" translates to choosing a value for each header field from one of the extrema of the range specified by each filter field. The `RandomCorner` function chooses a random "corner" of the filter identified by `RandFilt` and stores the header in `NewHeader`.

The last steps in the header generation loop append a variable number of copies of `NewHeader` to the trace. The number of copies, `Copies`, is chosen by sampling from a Pareto distribution controlled by the input parameters, *ParetoA* and *ParetoB* (line 10). In doing so, we provide a simple control point for the locality of reference in the header trace. The Pareto distribution[6] is one of the heavy-tailed distributions commonly used to model the burst size of Internet traffic flows as well as the file size distribution for traffic using the TCP protocol [35]. For convenience, let $a = ParetoA$ and $b = ParetoB$. The probability density function for the Pareto distribution may be expressed as:

$$P(x) = \frac{ab^a}{x^{a+1}} \tag{5}$$

where the cumulative distribution is:

$$D(x) = 1 - \left(\frac{b}{x}\right)^a \tag{6}$$

The Pareto distribution has a mean of:

$$\mu = \frac{ab}{a-1} \tag{7}$$

Expressed in this way, a is typically called the shape parameter and b is typically called the scale parameter, as the distribution is defined on values in the interval (b, ∞). The following are some examples of how the Pareto parameters are used to control locality of reference:

- Low locality of reference, short tail: ($a = 10$, $b = 1$) most headers will be inserted once
- Low locality of reference, long tail: ($a = 1$, $b = 1$) many headers will be inserted once, but some could be inserted over 20 times
- High locality of reference, short tail: ($a = 10$, $b = 4$) most headers will be inserted four times

Once the size of the trace exceeds the threshold, the header generation loop terminates. Note that a large burst near the end of the process will cause the trace to be larger than `Threshold`. After generating the list of headers, we write the trace to an output file (line 13).

7 Benchmarking with ClassBench

In order to provide value to the interested community, a packet classification benchmark must provide meaningful measurements that cover the broad spectrum of application environments. It is with this in mind that we designed the suite of *ClassBench* tools to be flexible while hiding the low-level details of filter set structure. While it is unclear if real filter sets will vary as specified by the smoothing and scope parameters, we assert that the tool provides a useful mechanism for measuring the effects of filter set composition on classifier performance. It is our intention to initiate and frame a broader discussion within the community that results in a larger set of *parameter files* that model real filter sets as well as the formulation of a standard benchmarking methodology.

[6]The Pareto distribution, a power law distribution named after the Italian economist Vilfredo Pareto, is also known as the Bradford distribution.

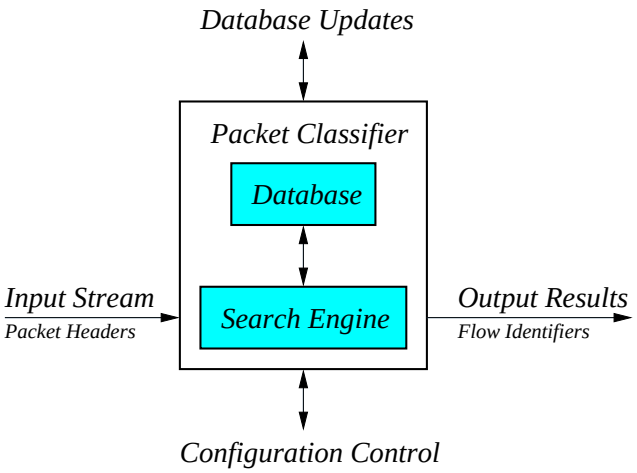


Figure 21: Generic model of a packet classifier.

Packet classification algorithms and devices range from purely conceptual, to software implementations targeted to a variety of platforms, to state-of-the-art ASICs (Application Specific Integrated Circuits). For the purpose of our discussion, we present a generic packet classifier model as shown in Figure 21. In this model, the classifier consists of a search engine connected to memory which stores the filter set and any other data structures required for the search. For each packet header passed to the classifier, the search engine queries the filter set and returns an associated flow identifier or set of flow identifiers. Note that the set of possible flow identifiers is application dependent. Firewalls may only specify two types of flows, admissible and inadmissible, whereas routers implementing per-flow queuing may specify thousands of unique flow identifiers. The configuration control is used to specify parameters such as the number of matching flow identifiers to return and the format of incoming packet headers. In order to model application environments where per-flow filters are dynamically created and deleted, the model includes a mechanism for dynamic filter set updates.

There are three primary metrics of interest for packet classification algorithms and devices: lookup throughput, memory requirements, and power consumption. Update performance is also a consideration, but secondary to the other three metrics. For packet classification devices or fixed implementations of algorithms, throughput can be directly measured using a synthetic filter set and associated header trace. Throughput measurements for software implementations of algorithms are not as straight-forward. In this case, the metric most directly influencing throughput is the required number of *sequential* memory accesses. Using parallel and pipelined design techniques, non-sequential memory accesses can be masked. A suitable benchmarking methodology should report both the total and sequential memory accesses in terms of average, worst observed, and best observed. The second metric of vital interest is the amount of memory required to store the filter set and supplemental data structures. For classification techniques employing random access memory, garnering memory usage metrics is straight-forward using a synthetic filter set. For TCAM-based devices, memory usage can be measured in terms of storage efficiency, which is defined to be the ratio of the number of required TCAM slots and the number of filters in the filter set.

In order to facilitate broader discussion, we make the *ClassBench* tools and 12 *parameter files* publicly available at the following site: `http://www.arl.wustl.edu/~det3/ClassBench/`. Input garnered from the community will be used to refine the tools suite, assemble a standard set of *parameter files*, and formally specify a benchmarking methodology. While we have already found *ClassBench* to be very useful in our own research, it is our hope to promote its broader use in the research community.

Acknowledgments

We would like to thank Ed Spitznagel for contributing his insight to countless discussions on packet classification and assisting in the "debugging" of the ClassBench tools.

References

[1] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," in *ACM Sigcomm*, August 1999.

[2] P. Gupta and N. McKeown, "Packet Classification using Hierarchical Intelligent Cuttings," in *Hot Interconnects VII*, August 1999.

[3] F. Baboescu and G. Varghese, "Scalable Packet Classification," in *ACM Sigcomm*, August 2001.

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

[5] E. Spitznagel, D. Taylor, and J. Turner, "Packet Classification Using Extended TCAMs," in *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, 2003.

[6] D. E. Taylor, "Survey & Taxonomy of Packet Classification Techniques," Tech. Rep. WUCSE-2004-24, Department of Computer Science & Engineering, Washington University in Saint Louis, May 2004.

[7] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis, "Framework for ip performance metrics." RFC 2330, May 1998.

[8] S. Bradner, "Benchmarking Terminology for Network Interconnect Devices." RFC 1242, July 1991.

[9] S. Bradner and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices." RFC 2544, March 1999.

[10] G. Trotter, "Terminology for Forwarding Information Base (FIB) based Router Performance." RFC 3222, December 2001.

[11] G. Trotter, "Methodology for Forwarding Information Base (FIB) based Router Performance." Internet Draft, January 2002.

[12] D. Newman, "Benchmarking Terminology for Firewall Performance." RFC 2647, August 1999.

[13] B. Hickman, D. Newman, S. Tadjudin, and T. Martin, "Benchmarking Methodology for Firewall Performance." RFC 3511, April 2003.

[14] P. Chandra, F. Hady, and S. Y. Lim, "Framework for Benchmarking Network Processors." Network Processing Forum, 2002.

[15] A. Feldmann and S. Muthukrishnan, "Tradeoffs for Packet Classification," in *IEEE Infocom*, March 2000.

[16] P. Warkhede, S. Suri, and G. Varghese, "Fast Packet Classification for Two-Dimensional Conflict-Free Filters," in *IEEE Infocom*, 2001.

[17] F. Baboescu and G. Varghese, "Fast and Scalable Conflict Detection for Packet Classifiers," in *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, 2002.

[18] T. Y. C. Woo, "A Modular Approach to Packet Classification: Algorithms and Results," in *IEEE Infocom*, March 2000.

[19] V. Sahasranaman and M. Buddhikot, "Comparative Evaluation of Software Implementations of Layer 4 Packet Classification Schemes," in *Proceedings of IEEE International Conference on Network Protocols*, 2001.

[20] Cisco, "CiscoWorks VPN/Security Management Solution," tech. rep., Cisco Systems, Inc., 2004.

[21] Lucent, "Lucent Security Management Server: Security, VPN, and QoS Management Solution," tech. rep., Lucent Technologies Inc., 2004.

[22] J. van Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 560–571, May 2003.

[23] M. E. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. T. Campbell, "Directions in Packet Classification for Network Processors," in *Second Workshop on Network Processors (NP2)*, February 2003.

[24] J. Postel, "Transmission Control Protocol." RFC 793, September 1981.

[25] J. Postel, "User Datagram Protocol." RFC 768, August 1980.

[26] J. Postel, "Internet Control Message Protocol." RFC 792, September 1981.

[27] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina, "General Routing Encapsulation." RFC 2784, March 2000.

[28] J. Moy, "OSPF Version 2." RFC 2784, July 1997.

[29] Cisco, "Enhanced Interior Gateway Routing Protocol (EIGRP)." white paper, 2003. Cisco Systems Inc.

[30] S. Kent and R. Atkinson, "IP Encapsulating Security Payload (ESP)." RFC 2406, November 1998.

[31] S. Kent and R. Atkinson, "IP Authentication Header." RFC 2402, November 1998.

[32] C. Perkins, "IP Encapsulation within IP." RFC 2003, October 1996.

[33] C. Bormann, et. al., "RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed." RFC 3095, July 2001. IETF Network Working Group.

[34] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *SIGCOMM 99*, pp. 135–146, 1999.

[35] Wikipedia, "Pareto distribution." Wikipedia, The Free Encyclopedia, April 2004. http://en.wikipedia.org/wiki/Pareto_distribution.