ClassBench: A Packet Classification Benchmark

David E. Taylor, Jonathan S. Turner
Applied Research Laboratory
Washington University in Saint Louis
{det3,jst}@arl.wustl.edu

***Abstract*— Packet classification is an enabling technology for next generation network services and often the primary bottleneck in high-performance routers. The performance and capacity of many algorithms and classification devices, including TCAMs, depend upon properties of the filter set and query patterns. Despite the pressing need, no standard filter sets or performance evaluation tools are publicly available. 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 tool suite also includes a *Trace Generator* that produces a sequence of packet headers to exercise packet classification algorithms with respect to a given filter set. Along with specifying the relative size of the trace, we provide a simple mechanism for controlling locality of reference. While we have already found *ClassBench* to be very useful in our own research, we seek to eliminate the significant access barriers to realistic test vectors for researchers and initiate a broader discussion to guide the refinement of the tools and codification of a formal benchmarking methodology. The *ClassBench* tools are publicly available at the following site:**
`http://www.arl.wustl.edu/˜det3/ClassBench/`

I. Introduction

DEPLOYMENT of next generation network services hinges on the ability of Internet infrastructure to perform packet classification at physical link speeds. A packet classifier must compare header fields of every incoming packet against a set of filters in order to assign a flow identifier that is used to apply security policies, application processing, and quality-of-service guarantees. 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 header fields comprising the standard IP 5-tuple. A packet classifier searches for the highest priority filter or set of filters matching the packet where each filter specifies a prefix of the IP source and destination addresses, an exact match or wildcard for the transport protocol number, and ranges for the source and destination port numbers for TCP and UDP packets.

As reported in Section III, 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]. Consequently, the performance of these approaches are subject to the structure of filter sets. Likewise, the capacity and efficiency of the most prominent packet classification solution, Ternary Content Addressable Memory (TCAM), is also subject to the characteristics of filter sets [1]. Despite the influence of filter set composition on the performance of packet classification search techniques and devices, no publicly available benchmarking tools or filter sets exist for standardized performance evaluation. Due to security and confidentiality issues, access to large, real filter sets 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 a small set of high-level inputs, and 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 IV we discuss the statistics and probability distributions contained in the *parameter files* that drive the synthetic filter generation process.

The *Filter Set Generator* takes as input a *parameter file* and a few high-level parameters. In addition the filter set *size* parameter, the *smoothing* and *scope* parameters provide high-level control over the composition of the filter set, abstracting the user 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 V. Finally, the *Trace Generator* tool examines the syn-

This work supported by the National Science Foundation, ANI-9813723.

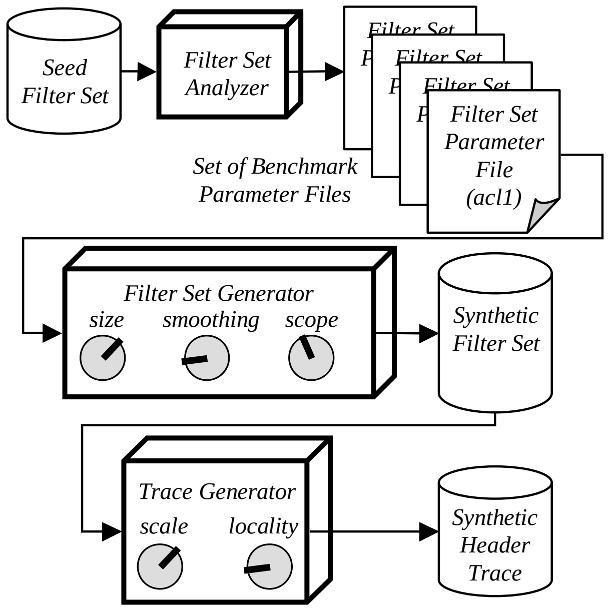


Fig. 1. Block diagram of the *ClassBench* tool suite. The synthetic *Filter Set Generator* has size, smoothing, and scope adjustments which provide high-level, systematic mechanisms for altering the size and composition of synthetic filter sets. The set of benchmark *parameter files* model real filter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

thetic 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 VI.

We highlight previous performance evaluation efforts by the research community as well as related benchmarking activity of the IETF in Section II. 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:

- *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.

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/`

II. 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. 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 [2]. The Benchmarking Methodology Working Group (BMWG) seeks to make measurement recommendations for various internetworking technologies [3]. 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 [4] and also produced a methodology for benchmarking firewalls [5]. 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. *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 [6]. 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 packet classification.

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 [7]. Feldmann and Muthukrishnan composed filter sets based on *NetFlow* packet traces from commercial networks [8]. Several groups have generated synthetic two-dimensional filter sets consisting of source-destination address prefix pairs by randomly selecting address prefixes from publicly available route tables [9], [8], [10]. 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 [11], [12]. 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 [11]. Baboescu and Varghese also introduced a scheme for using a sample filter set to generate a larger synthetic five-dimensional filter set [13]. 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 filter set composition which is crucial for evaluating algorithms that exploit filter set characteristics.

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.) [14]. 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 [15].

III. 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 performance. The focus of this section is to identify and understand the impetus for the observed structure of filter sets 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), Firewall (FW), and IP Chain (IPC). 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. Nonetheless, the following analysis provide useful insight into the structure of real filter sets. We observe that various useful properties hold regardless of filter set size or format. Due to space constraints, we are unable to fully elaborate on our analysis, but a more complete discussion of this work is available in technical report form [16].

A. 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) [17] and Lucent Security Management Server (LSMS) [18]. 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.

TABLE I
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.

Port	*WC*	*HI*	*LO*	*AR*	*EM*
Source	78.08	6.60	0.92	0.42	13.99
Destination	40.39	6.18	0.06	4.33	49.04

A set of applications may be identified by specifying ranges for the source and destination port numbers.

B. 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.

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. Filters specified one of nine protocols or the wildcard. The most common protocol specification was TCP (49%), followed by UDP (27%), the wildcard (13%), and ICMP (10%). The following protocols were specified by less than 1% of the filters: General Routing Encapsulation (GRE), Open Shortest Path First (OSPF) Interior Gateway Protocol (IGP), Enhanced Interior Gateway Routing Protocol (EIGRP), IP Encapsulating Security Payload (ESP) for IPv6, IP Authentication Header (AH) for IPv6, IP Encapsulation within IP (IPE).

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 $[1024 : 65535]$
- LO, well-known system port range $[0 : 1023]$
- AR, arbitrary range
- EM, exact match

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. We computed the distribution of filters over the five classes for both source and destination ports for each filter set. Table I shows the combined distribution for all filter sets. We observe some interesting trends in the raw 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. Only one filter set contained filters specifying the LO port class for either the source or destination port range.

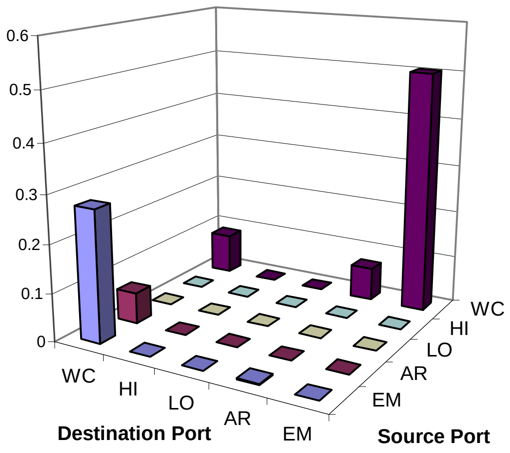


Fig. 2. *Port Pair Class Matrix* for TCP, filter set fw4.

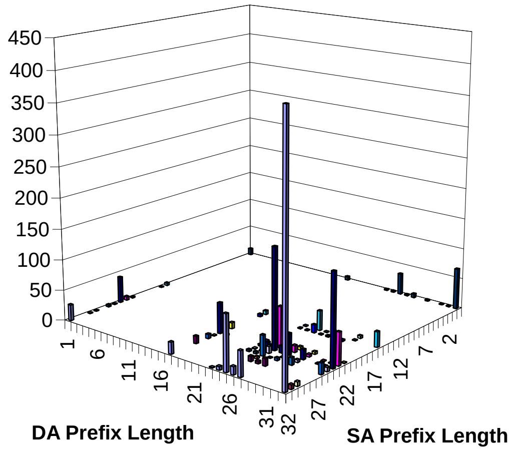


Fig. 3. Prefix length distribution for address prefix pairs in filter set ipc1.

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 examined the *PPC Matrix* defined by filters specifying the same protocol. For all protocols except TCP and UDP, the *PPC Matrix* is trivial – a single spike at WC/WC. Figure 2 shows the *PPC Matrix* defined by filters specifying the TCP protocol in filter set fw4.

C. 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 classification algorithms depend upon the number of unique prefix lengths and the distribution of filters across those unique values. We find that a majority of the filter sets specify fewer than 15 unique prefix lengths for either source or destination address prefixes. The number of unique source/destination prefix pair lengths is typically less than 32, which is small relative to the filter set size and the number of possible combinations, 1024. For example, the largest filter set contained 4557 filters, 11 unique source address prefix lengths, 3 unique destination address lengths, and 31 unique source/destination prefix pair lengths.

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. 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 filter set *ipc1* shown in Figure 3. There are very few filters specifying a 24-bit prefix for either the source or destination address, a notable difference from backbone route tables which are dominated by class C address prefixes (24-bit network address) and their aggregates. Finally, we observe that while the distributions for different filter sets 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. 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. Consider a 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 conditional 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. As shown in Figure 4, $p\{0|11\}$ is the probability that the 0 path is chosen at level 2 given that the 1 path was chosen at level 0 and the 1 path was chosen at level 1. Such a characterization is overly complex, hence we employ suitable metrics that capture the important characteristics while providing a more concise representation.

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 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 a 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

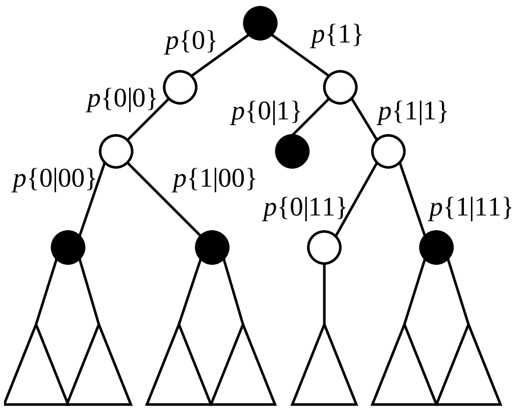


Fig. 4. Example of complete statistical characterization of address prefixes.

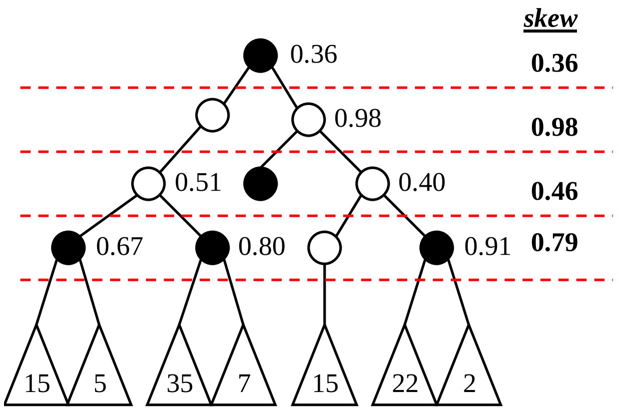
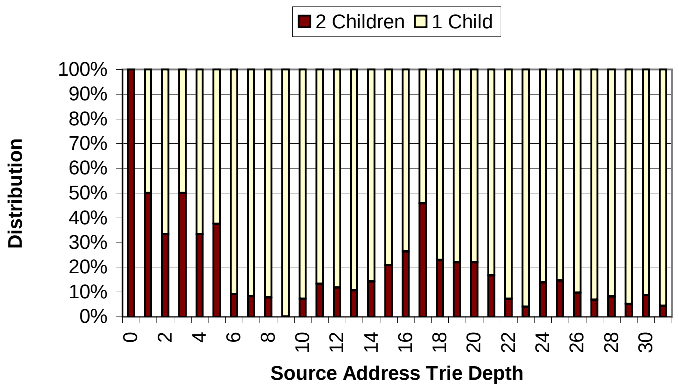
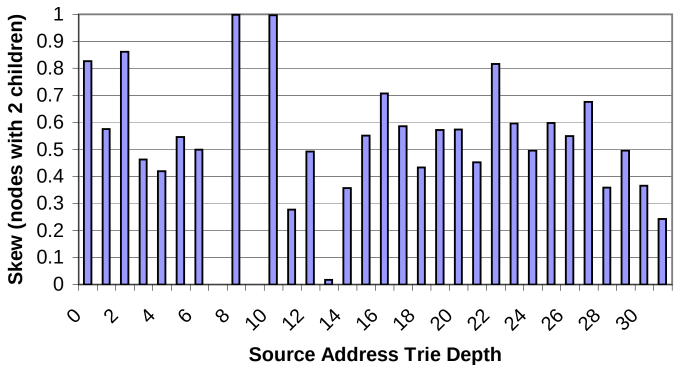


Fig. 5. Example of skew computation for the first four levels of an address trie; shaded nodes denote a prefix specified by a single filter; subtrees denoted by triangles with associated weight.



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



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

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

and let $light$ be the subtree with equal or less weight, thus:

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

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. Finally, this definition of skew provides an anonymous measure of address prefix structure, as it does not preserve address prefix values.

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 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. Additional details regarding the "correlation" metric and results from real filter sets may be found in the technical report [16].

D. Scope

Next we seek to characterize the *specificity* of the filters in the filter set. 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. The number of possible packet headers covered by a filter is characterized by its *tuple* specification. To be specific, we consider the standard 5-tuple as a vector containing the following fields:

- $t[0]$, 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, $[0, 1]$

We define a new metric, *scope*, to be the logarithmic measure of the number of possible packet headers covered by the filter. Using the definition above, we define a filter's 5-tuple *scope* as follows:

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

Thus, *scope* is a measure of filter specificity on a scale from 0 to 104. The average 5-tuple scope for our 12 filter sets ranges from 56 to 24. We note that filters in the ACL filter sets tend to have narrower scope, while filters in the FW filter sets tend to have wider scope.

E. 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 filter set *ipc1* contain non-wildcard flags. We argue that new services and administrative policies will demand that packet classification techniques scale to support additional fields beyond the standard 5-tuple. Matches on ICMP type number and other higher-level header fields are likely to be exact matches. There may be other types of matches that more naturally suit the application, such as arbitrary bit masks on TCP flags.

IV. Parameter Files

Given a real filter set, the *Filter Set Analyzer* generates a *parameter file* that contains statistics and probability distributions that allow the *Filter Set Generator* to produce a synthetic filter set that retains the relevant characteristics of 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. Note that *parameter files* also provide complete anonymity of addresses in the original filter set. By reducing confidentiality concerns, we seek to remove the significant access barriers to realistic test vectors for researchers and promote the development of a benchmark set of *parameter files*. There still exists a need for a large sample space of real filter sets from various application environments. We have generated a set of 12 *parameter files* which are publicly available along with the *ClassBench* tool suite.

Parameter files include the following entries[1]:

- *Protocol* specifications and the distribution of filters over those values
- *Port Pair Class Matrix* for each unique protocol specification in the filter set
- *Flags* specifications for each protocol and a distribution of filters over those values
- *Arbitrary port range* specifications and a distribution of filters over those values for both the source and destination port fields
- *Exact port number* specifications and a distribution of filters over those values for both the source and destination port fields
- *Prefix pair length* distribution for each *Port Pair Class Matrix*
- *Address prefix branching and skew* distributions for both source and destination address prefixes
- *Address prefix correlation* distribution
- *Prefix nesting thresholds* for both source and destination address prefixes.

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[2] as shown in Figure 7. 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 V-B, 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.

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 V.

[1] We avoid an exhaustive discussion of *parameter file* contents and format details; interested readers and potential users of *ClassBench* may find a discussion of *parameter file* format in the documentation provided with the tools.

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

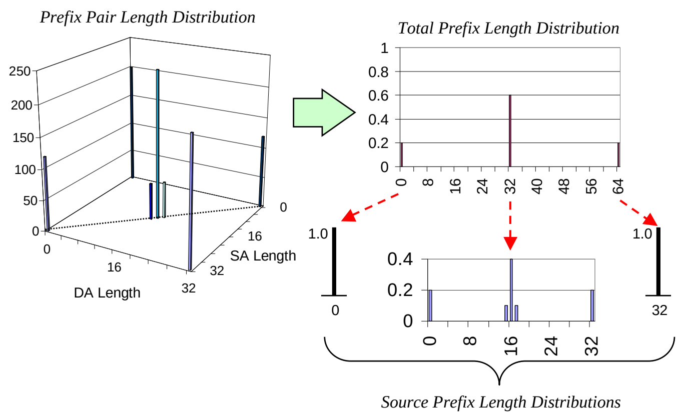


Fig. 7. *Parameter files* represent prefix pair length distributions using a combination of a total prefix length distribution and source prefix length distributions for each non-zero total length.

V. Synthetic Filter Set Generation

The *Filter Set Generator* is the cornerstone of the *ClassBench* tool suite. Perhaps the most succinct way to describe the synthetic filter set generation process is to walk through the pseudocode shown in Figure 8. The first step in the filter generation process is to read the statistics and distributions from the *parameter file*. 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[3](lines 6 through 10). 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). We discuss this adjustment and its effects on the generated filter set in Section V-A.

The next set of steps (lines 12 through 27) generate a *partial* filter for each entry in the `Filters` array. Essentially, 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 (line 14) using a uniform random variable, `rv` (line 13). We chose to select the protocol first because we found that the protocol specification

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

Fig. 8. Pseudocode for *Filter Set Generator*.

dictates the structure of the other filter fields. Next, we select the protocol flags[4] from the `Flags` distribution associated with the chosen protocol (line 16).

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). 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 *PPCs*, 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 V-B. Given the *PPC*, 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.

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 *PPC* (line 24) using a random variable biased by the *address scope* parameter (line 23). We select a source prefix length from the distribution associated with the chosen *PPC* and total length (line 26) using a uniform random variable (line 25). 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. The `AssignSA` routine recursively 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* (line 28). The recursive process first examines all of the entries in `FilterList`. If an entry has a source prefix length equal to the level of the node, it assigns the node's address to the entry and removes the entry from `FilterList`. The process then 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 and ensure that the prefix 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 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. Additional details regarding the address prefix assignment process are included in the technical report [16].

[4]Note that the protocol flags field is typically the wildcard unless the chosen protocol is TCP or ICMP.

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 V-C. 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 V-C.

A. Smoothing Adjustment

As filter sets scale in size, we anticipate that new address prefix pair lengths will emerge due to subnet 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 during the generation process neglects the structure of the filter set used to generate the *parameter file*. Using scope as a measure of distance, subnet aggregation and segregation results in new prefix lengths that are "near" to the original prefix length. Consider the address prefix pair length distribution where all filters in the filter set have 16-bit source and destination address prefixes; thus, the distribution is a single "spike". In order to model aggregation and splitting of subnets, new prefix pair lengths should 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

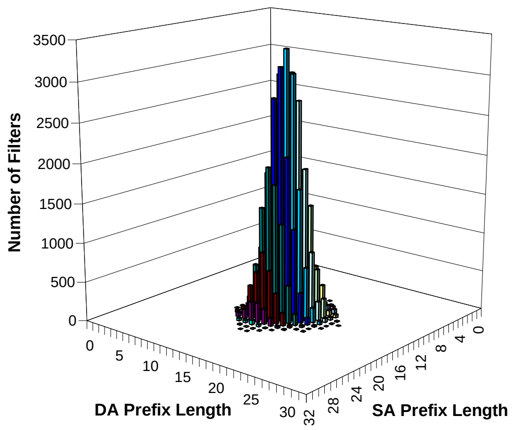


Fig. 9. Prefix pair length distributions for a synthetic filter set of 64000 filters generated with a *parameter file* specifying 16-bit prefix lengths for all addresses and smoothing parameter $r = 8$.

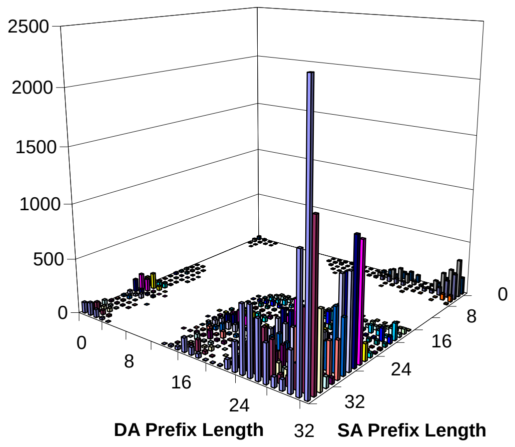


Fig. 10. Prefix pair length distribution for a synthetic filter set of 64000 filters generated with the ipc1 *parameter file* with smoothing parameter $r = 4$.

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:

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

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.

In order to demonstrate this process, Figure 9 shows the prefix pair length distribution for a synthetic filter set generated with a *parameter file* specifying 16-bit prefix lengths for all addresses and a smoothing parameter $r = 8$. In practice, we expect that the *smoothing* parameter will be limited to at most 8. In order to demonstrate the effect of smoothing on a real filter set, Figure 10 shows 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 shown in Figure 3 while modeling a realistic amount of address prefix aggregation and segregation.

B. 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 average scope decreases. If the number of explicitly blocked ports for all packets in a firewall filter set increases, then the average scope may increase[5]. In order to explore the performance effects of filter scope, 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 sample from a cumulative distribution, we typically choose a random number uniformly distributed between zero and one, rv_{uni}, then chooses the value covering rv_{uni} in the cumulative distribution. Graphically, this amounts to projecting a horizontal line from the random number on the *y*-axis. The chosen value is the *x*-coordinate of the intersection of the cumulative distribution and the *y*-projection of the random number. In Figure 11, we shown an example of sampling from a cumulative total prefix pair length distribution with $rv_{uni} = 0.5$ to choose the total prefix pair length of 44. The *scope* adjustments bias the sampling process to select more or less specific *Port Pair Classes* and 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. Consider the case of selecting prefix pair lengths. The first option requires that we recompute the cumulative distribution to make longer or shorter total 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 11, 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}.

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

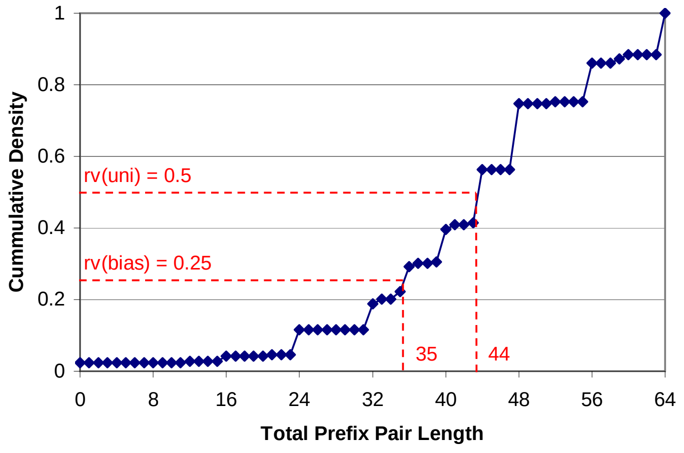


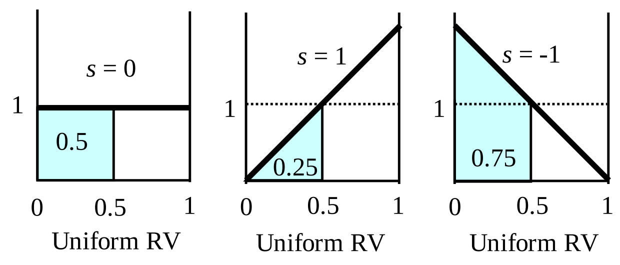
Fig. 11. Example of sampling from a cumulative distribution using a uniform random variable, and a biased random variable. Distribution is for the total prefix pair length associated with the WC-WC port pair class of the acl2 filter set.

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 12(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 to the uniform random variable. We can bias the random variable by altering the slope of the line. In order for the biasing function to have a range of $[0 : 1]$ for random variables in the range $[0 : 1]$, the slope adjustment must be in the range $[-2 : 2]$. 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 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:

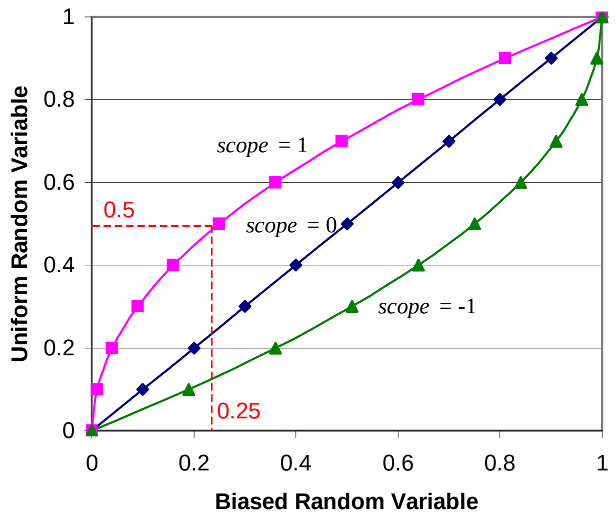
$$rv_{bias} = rv_{uni}(scope \times rv_{uni} - scope + 1) \quad (4)$$

Figure 12(b) shows a plot of the biasing function for *scope* values of 0, -1, and 1, as well as an 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 11, we also show an example of sampling the distribution using the biased random variable, $rv_{bias} = 0.25$, resulting from applying the biasing function with $scope = 1$. 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



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



(b) Plot of scope biasing function.

Fig. 12. Scope applies a biasing function to a uniform random variable.

of the filter set. The same effects are realized by the *port scope* adjustment by biasing the *Filter Set Generator* to select more or less specific *Port Pair Classes*.

Finally, we show 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* in Figure 13. Each data point in the plot is from a synthetic filter set containing 16000 filters generated from a *parameter file* from filter sets acl3, fw5, or ipc1. For these tests, both scope parameters were set to the same value. Over their range of values, the scope parameters alter the average filter scope by ± 6 to ± 7.5. We also measured the individual effects of the *address scope* and *port scope* parameters. Over its range of values, the *address scope* alters the average address pair scope by ± 4 to ± 6. Over its range of values, the *port scope* alters the average port pair scope by ± 1.5 to ± 2.5. 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.

C. *Filter Redundancy & Priority*

The final steps in synthetic filter set generation are removing redundant filters and ordering the remaining filters in order of

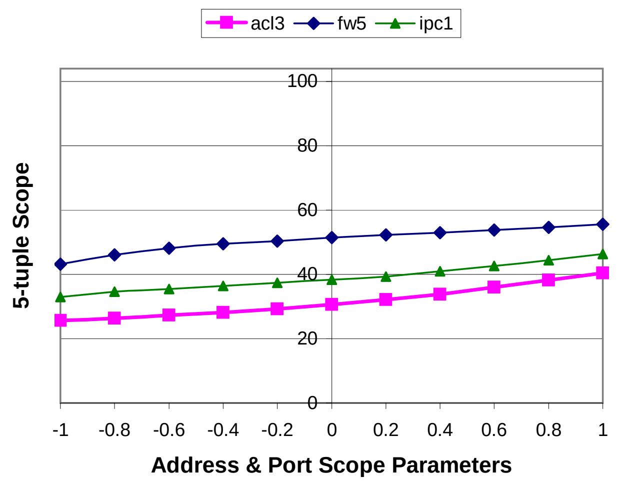


Fig. 13. Average scope of synthetic filter sets consisting of 16000 filters generated with parameter files extracted from filter sets *acl3*, *fw5*, and *ipc1*, and various values of the scope parameters.

```
TraceGenerator()
   // Generate list of synthetic packet headers
1  read($FilterSet$)
2  get($scale$)
3  get($ParetoA$)
4  get($ParetoB$)
5  Threshold = $scale \times$ 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$\rightarrow$append(NewHeader)
13 Headers$\rightarrow$print
```

Fig. 14. Pseudocode for *Trace Generator*.

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. Such an approach makes execution times of the *Filter Set Generator* prohibitively long for filter sets with more than a few thousand filters. In order to accelerate this process, we first sort the filters into sets according to their tuple specification. 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 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.

VI. 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 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 14.

We begin by reading the *FilterSet* (line 1) and getting 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. The next set of steps continue to generate synthetic headers as long as the size of `Headers` does not exceed the `Threshold` defined by the product of *scale* and the number filters in *FilterSet*.

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. 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 [19]. 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).

VII. Benchmarking with ClassBench

We have already found *ClassBench* to be tremendously valuable in our own research [20]. In order to provide value for the broader community, a packet classification benchmark must provide meaningful measurements that cover the 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 believe that the tool provides a useful mechanism for measuring the effects of filter set composition on classifier performance. It is our hope that *ClassBench* will enjoy broader use by researchers in need of realistic test vectors; it is also 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.

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. We also would like to thank Venkatachary Srinivasan and Will Eatherton for making real filter sets available for study.

References

[1] 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.
[2] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis, "Framework for ip performance metrics." RFC 2330, May 1998.
[3] S. Bradner and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices." RFC 2544, March 1999.
[4] G. Trotter, "Methodology for Forwarding Information Base (FIB) based Router Performance." Internet Draft, January 2002.
[5] B. Hickman, D. Newman, S. Tadjudin, and T. Martin, "Benchmarking Methodology for Firewall Performance." RFC 3511, April 2003.
[6] P. Chandra, F. Hady, and S. Y. Lim, "Framework for Benchmarking Network Processors." Network Processing Forum, 2002.
[7] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," in *ACM Sigcomm*, August 1999.
[8] A. Feldmann and S. Muthukrishnan, "Tradeoffs for Packet Classification," in *IEEE Infocom*, March 2000.
[9] P. Gupta and N. McKeown, "Packet Classification using Hierarchical Intelligent Cuttings," in *Hot Interconnects VII*, August 1999.
[10] P. Warkhede, S. Suri, and G. Varghese, "Fast Packet Classification for Two-Dimensional Conflict-Free Filters," in *IEEE Infocom*, 2001.
[11] F. Baboescu and G. Varghese, "Scalable Packet Classification," in *ACM Sigcomm*, August 2001.
[12] F. Baboescu and G. Varghese, "Fast and Scalable Conflict Detection for Packet Classifiers," in *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, 2002.
[13] F. Baboescu, S. Singh, and G. Varghese, "Packet Classification for Core Routers: Is there an alternative to CAMs?," in *IEEE Infocom*, 2003.
[14] T. Y. C. Woo, "A Modular Approach to Packet Classification: Algorithms and Results," in *IEEE Infocom*, March 2000.
[15] 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.
[16] D. E. Taylor and J. S. Turner, "ClassBench: A Packet Classification Benchmark," Tech. Rep. WUCSE-2004-28, Department of Computer Science & Engineering, Washington University in Saint Louis, May 2004.
[17] Cisco, "CiscoWorks VPN/Security Management Solution," tech. rep., Cisco Systems, Inc., 2004.
[18] Lucent, "Lucent Security Management Server: Security, VPN, and QoS Management Solution," tech. rep., Lucent Technologies Inc., 2004.
[19] Wikipedia, "Pareto distribution." Wikipedia, The Free Encyclopedia, April 2004. http://en.wikipedia.org/wiki/Pareto_distribution.
[20] D. E. Taylor and J. S. Turner, "Scalable Packet Classification using Distributed Crossproducting of Field Labels," Tech. Rep. WUCSE-2004-38, Department of Computer Science and Engineering, Washington University in Saint Louis, June 2004.