Packet Classification using Hierarchical Intelligent Cuttings

Pankaj Gupta and Nick McKeown
Computer Systems Laboratory, Stanford University
Stanford, CA 94305-9030
{pankaj, nickm}@stanford.edu

Abstract

Internet routers that operate as firewalls, or provide a variety of service classes, perform different operations on different flows. A flow is defined to be all the packets sharing common header characteristics; for example a flow may be defined as all the packets between two specific IP addresses. In order to classify a packet, a router consults a table (or classifier) using one or more fields from the packet header to search for the corresponding flow. The classifier is a list of rules that identify each flow and the actions to be performed on each. With the increasing demands on router performance, there is a need for algorithms that can classify packets quickly with minimal storage requirements and allow new flows to be frequently added and deleted. In the worst case, packet classification is hard requiring routers to use heuristics that exploit structure present in the classifiers. This paper presents such a heuristic, called *HiCuts*, (hierarchical intelligent cuttings), which exploits the structure found in classifiers. We describe *HiCuts* and examine its performance against real classifiers in use today. When compared with previously described algorithms and used to classify packets based on four header fields, the algorithm is found to classify packets quickly and has relatively small storage requirements.

1 Introduction

Packet classification is employed by Internet routers to implement a number of advanced Internet services, such as routing, rate limiting, access-control in firewalls, virtual bandwidth allocation, policy-based routing, service differentiation, load balancing, traffic shaping, and traffic billing. Each of these services require the router to classify incoming packets into different flows and then perform appropriate actions depending upon which flow the incoming packet has been identified to fall into. These flows, or classes, are specified by a classifier. A classifier is a set of *filters* or *rules*. For instance, each rule in a firewall could specify a set of source and destination addresses, and associate a corresponding 'deny' or 'permit' action with it. Or the rules could be based on several fields of the packet including layer 2, 3, 4 and may be 5 addressing and protocol information.

The simplest, and most well-known form of packet classification is used in routing IP datagrams, where each rule specifies a destination prefix. The associated action is the IP address of the next-hop where the packet needs to be routed to. The classification process requires determining the longest prefix which matches the destination address of the packet.

2 Generic Packet Classification

Generic packet classification requires the router to classify a packet based on multiple fields in its header. Each rule of the classifier specifies a *class*† that a packet may belong to based on some criteria on F fields of the packet header, and associates with each class an identifier, *classID*. This identifier uniquely specifies the action associated with the rule. Each rule has F components. The i^{th} component of rule R, referred to as $R[i]$, is a regular expression on the i^{th} field of the packet header.‡ A packet P is said to *match* a particular rule R, if $\forall i$, the i^{th} field of the header of P satisfies the regular expression $R[i]$.

The classes specified by the rules may be overlapping i.e. one packet can match several rules. Without loss of generality, we will assume throughout this paper that when

† For example, each rule in a flow classifier is a flow specification, where each flow is in a separate class.

‡ In practice, the regular expression is limited by syntax to a simple address/mask or operator/number(s) specification.

two rules overlap, the order in which they appear in the classifier will determine their relative priority. In other words, a packet which matches multiple rules will belong to the class identified by the *classID* of the rule *R*, if *R* is the *first* among all the rules the packet matches in the classifier.

3 Related Work

The simplest classification algorithm is a linear search of each rule of a classifier. Of course, for large classifiers this approach requires a long query time, but is very efficient in terms of storage requirements. The data structure is simple and is readily updated as rules change.

A ternary CAM (content addressable memory) is a hardware device performing the function of a fully associative memory. A cell in a ternary CAM can store three values: '0', '1' or 'X'. The 'X' value represents a don't care and operates as a per-cell mask enabling the ternary CAM to match rules containing wildcards. In terms of its operation, a ternary CAM seems almost ideally suited to packet classification: one can present a whole packet header to the device and determine which entry (or entries) it matches. However, the complexity of CAMs has traditionally only permitted small, inflexible and relatively slow implementations that consume a lot of power. But although ternary CAM technology is not quite ready today, improvements in semiconductor technology might make dense, fast ternary CAMs feasible in the future. In the meantime, there is still a need for efficient algorithmic solutions operating on specialized data structures.

One such algorithm for two dimensions, called *Grid of Tries*, is proposed by Srinivasan et al. [6]. In this scheme, a trie data structure is extended to two fields. Srinivasan et al. show that on a classifier with 20,000 rules in two dimensions, the scheme requires about 9 memory accesses per query in the worst case and about 2MB for storage. The authors found that the scheme cannot be easily extended to more than two fields, and so proposed a generalized scheme called '*Crossproducting*'. They show that the scheme works well with classifiers smaller than about 50 rules but requires caching for larger classifiers.

An alternative hardware-optimized scheme using *bit-level parallelism* is proposed by Lakshman and Stiliadis [5]. The authors show fast query times and small storage requirements for small classifiers but the storage requirement is found to scale quadratically and the memory bandwidth linearly with the size of the classifier, making the scheme impractical for large classifiers. Variations are proposed by the authors that optimize under certain conditions (e.g. decreasing the storage requirement at the expense of longer query time; or optimizing for lookups in two fields). The scheme does not appear to work well for large multi-dimensional classifiers.

More recently, we proposed a packet classification algorithm, called *Recursive Flow Classification* for classification on multiple fields[2]. We showed that it works well for real-life classifiers, but the storage requirements are still high (up to 3MB). Also, we have as yet been unable to provide a mechanism for doing incremental updates to the data structure.

Another algorithm called *Tuple-Space Search* has been recently proposed for packet classification on multiple fields [3]. The scheme partitions the rules of a classifier into different tuple categories based upon the number of specified bits in the rules (a bit is specified in a rule if it is not a "don't care" bit). The scheme then uses hashing among rules within the same category. The main advantages of this algorithm are its fast average query time and fast update time when rules change. The main disadvantage is the use of hashing which leads to lookups or updates of non-deterministic duration.

It is possible to find worst-case bounds on the complexity of queries and storage requirements by casting the packet classification as a problem in computational geometry. In particular, the "point location problem" (where one has to find the enclosing region of a query point, given a set of non-overlapping regions) can be reduced to the problem of packet classification. For n non-overlapping regions in F dimensions (fields), two well known results trade-off query time against storage requirements. Optimizing for query time, it is possible to achieve $O(\log n)$ complexity for query times, but with a complexity of $O(n^F)$ in storage; while optimizing for storage leads to $O(n)$ storage requirements, but $O(\log^{F-1} n)$ for queries [4]. Clearly, both extremes are impractical: with just 1000 rules and 3 fields, n^F storage is about 1GB; and $\log^{F-1} n$ time is about 100 memory accesses. Moreover the constants are big, and hence the techniques are of little practical significance. Also, the algorithms and data structures (not described here) can not be generalized in a straightforward manner to the case of overlapping regions.

We can draw two conclusions from the algorithms above: (1) The theoretical bounds tell us that it is not possible to arrive at a practical worst-case solution. Fortunately, we don't have to. Real-life classifiers have some inherent structure which it appears to be possible to exploit using simple heuristics. (2) No single algorithm will perform well for all cases, e.g. a simple and memory-efficient linear search, or the hardware solution in [5], might be sufficient when the number of filters is small. Hence a hybrid scheme might be able to combine the advantages of several different approaches.

In this paper we focus on practical implementation of classification on real-life classifiers. We present an approach which attempts to partition in each dimension, guided by simple heuristics to exploit the structure of the classifier. This structure is discovered by preprocessing the classifier. Parameters of the algorithm can be tuned to trade-off query time against storage requirements. Our results suggest that a classifier with 20,000 rules in two dimensions (where one dimension of the rules is generated randomly while the other dimension is taken from publicly available routing tables) consumes about 1.3MB of storage with a worst case query time of 4 memory accesses (and an average case of 2.3 memory accesses) plus a linear search on a small number of rules (four). On 40 real-life four-dimensional classifiers obtained from ISP and enterprise networks with 100 to 1700 rules, *HiCuts* requires less than 1MB of storage with a worst case query time of 12 and average case query time of 8 memory accesses, plus a linear search on eight rules. The preprocessing time can be sometimes large, nearly a minute, but fortunately the time to update a rule in the data structure is less than 0.1 seconds.

4 Packet Classification using Hierarchical Intelligent Cuttings (*HiCuts*)

The HiCut algorithm works by carefully preprocessing the classifier to build a decision tree data structure. Each time a packet arrives, the decision tree is traversed to find a leaf node, which stores a small number of rules. A linear search among these rules yields the desired matching. The shape and depth of the decision tree as well as the local decisions to be made at each node in the tree are chosen when the search tree is built.

With each internal node v of a k-dimensional classifier, we associate:

- A box $B(v)$, which is a k-tuple of ranges or intervals: $([l_1:r_1], [l_2:r_2],..., [l_k:r_k])$.

- A cut *C(v)*. The cut *C(v)* is defined by a dimension *d*, and *np(C)*, the number of times that box *B(v)* is cut (or partitioned†) in dimension *d* (i.e. cuts in the interval $[l_d:r_d]$). The cut thus divides *B(v)* into smaller boxes which are then associated with the children of *v*.
- A set of rules, *R(v)*. The root of the tree has all the rules associated with it. If *u* is a child of *v*, then *R(u)* is defined to be the subset of *R(v)* that collides with *B(u)*, i.e. every rule in *R(v)* that spans, cuts or is contained in *B(u)* is also a member of *R(u)*. We call *R(u)* the *CollidingRuleSet* of *u*.

As an example, consider the case of two *w*-bit wide dimensions. The root node represents a box of size $2^w \times 2^w$. The cuttings are made by axis-parallel hyperplanes (which are just lines in two dimensions). The cut *C* is described by the number of equal-sized intervals that a particular dimension of the box *B* is cut into. If we decide to cut the root node along the first dimension into *D* intervals, the root node will have *D* children, each with a box of size $\left(2^w/D\right) \times 2^w$ associated with it.

The process of cutting is performed at each level, and recursively on the child nodes of that level, until the number of rules in the box associated with each node fall below a threshold (which we will refer to as *binth*). A node with fewer than *binth* rules is not partitioned further and becomes a leaf of the tree. To illustrate this process, an example classifier is shown in Table 1. The same classifier is illustrated geometrically in Figure 1. The decision tree made by recursively cutting is shown in Figure 2 (in the example, *binth = 2)*.

† *Cut* and *Partition* are used synonymously throughout this paper.

Table 1:

Rule	Xrange	Yrange
R1	0-31	0-255
R2	0-255	128-131
R3	64-71	128-255
R4	67-67	0-127
R5	64-71	0-15
R6	128-191	4-131
R7	192-192	0-255

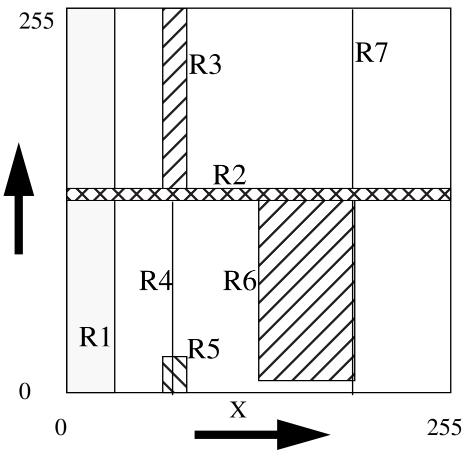
For any given classifier, there are possibly many ways to construct a decision tree, and so we guide the preprocessing using some heuristics based on structure present in the classifier. When performing cuts on node *v*:

1) The preprocessing algorithm uses a heuristic to pick a suitable *np(C)* i.e. the number of interval cuts to make. A large value for *np(C)* will decrease the depth of the tree (which will accelerate query time) at the expense of increasing storage. To balance this trade-off, the heuristic we follow is guided and tuned by a pre-determined space measure function *spmf()*. Let *NumRules(u)* = cardinality of *CollidingRuleSet(u)*. For a cut *C*, define a space measure,

$$sm(C(v)) = \sum_i NumRules(\text{child}_i) + np(C(v)).$$

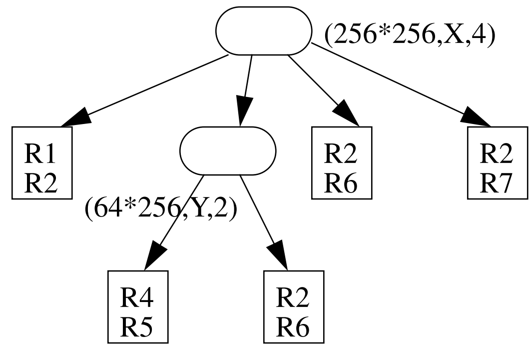
We make as many cuttings as the *spmf()* function allows us to at a certain node depending upon the number of rules at that node. This is done by a simple binary search on the number of cuttings till $sm(C(v))$ gets "close" enough to *spmf(NumRules(v))*. An algorithm for doing this search is shown in the Appendix 8.1.

Figure 1: An example classifier in two dimensions with 7 filters (w=8).



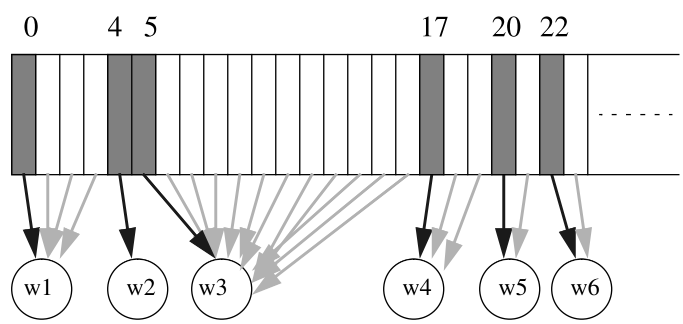
2) The preprocessing algorithm uses a heuristic to pick which dimension to cut along at each internal node. For example, it can be seen from Figure 1 that cutting the Y-axis would be less beneficial than cutting the X-axis. There are various metrics that can be used to pick the dimension, including for example: *(a)* Minimizing $max_j(NumRules(child_j))$ in an attempt to decrease the worst-case depth of the tree; *(b)* Treating $NumRules(child_j)/sm(C)$ as a probability distribution with *np(C)* elements, and maximizing the entropy of the distribution. Intuitively, this attempts to pick a dimension that leads to the most uniform distribution of rules across nodes; *(c)* Minimize *sm(C)* over all dimensions; *(d)* Cut the dimension that has the largest number of distinct components of rules in that dimension. In our example, *R3* and *R5* share the same rule component (range) in the X dimension.

Figure 2: A possible tree with binth=2 for the example classifier in Figure 1. Each ellipse denotes an internal node v with a triplet *(B(v), dim(C(v)), np(C(v)))*. Each square is a leaf node which contains the actual rules.



3) The preprocessing algorithm uses a heuristic to maximize the reuse of child nodes. We have observed that in real classifiers many child nodes have identical *CollidingRuleSets*. Hence, we can use a single child node for each distinct *CollidingRuleSet* and have identical child nodes point to it.

Figure 3: An example of using a heuristic to maximize the reuse of child nodes. The shaded regions correspond to children with distinct *CollidingRuleSets*.



4) The preprocessing algorithm uses a heuristic to eliminate redundancies in the tree. After some rules are cut, they might become redundant, i.e. the cut portion might become covered by a higher priority rule. In our example, if *R6* were higher priority than *R2*, then *R2* would be made redundant by *R6* in the third child of the root node. Detection and elimination of these

redundant rules can decrease the storage requirements, but is time consuming. In our experiments, we have only invoked this heuristic when the number of rules at a node has fallen below a threshold.

5 Implementation Results

To test how well the *HiCut* algorithm works with real and synthesized classifiers, we built a simple simulator. In the results that follow, we examine, for each classifier, the number of memory references (as a measure of query time) and the storage requirements. For each classifier, a search tree is built using the heuristics described above. The preprocessing algorithm is tuned by two parameters: (1) *binth*, and (2) *spfac* — used in the function *spmf()* defined as *spmf(N) = spfac * N*.

5.1 Two Dimensions

Two dimensional classifiers were created by picking values (prefixes) in both dimensions at random from publicly available routing tables [7]. Wildcards was also added at random to each dimension. To illustrate the results: for *binth=4*, a classifier with 20,000 rules was found to consume about 1.3MB of memory with a tree depth of 4 in the worst case and 2.3 on the average. The complete set of results are omitted here as they were found to be very similar to the results for higher-dimension classifiers described below.

5.2 Higher Dimensions

For more than two dimensions, about 40 classifiers containing between 100 and 1733 rules were taken from real ISP and enterprise networks†. These classifiers were used as access control lists in firewalls and had fields in four dimensions: source IP address, destination IP address, layer four protocol and layer four destination port. Further details about the characteristics of these classifiers is described in [2].

Figure 4 shows the total storage requirements for the classifiers (for the tree data structure and the classifier itself) when $binth = 8$ and $spfac = 4$. As can be seen, the maximum value of space consumed for any classifier is about 1Megabyte with the second highest less than 500Kilobytes. These small storage requirements mean that the data structure could readily fit in the L2 cache of most CPUs today.

Figure 4: Storage requirements for four dimensional classifiers for binth=8 and spfac=4.

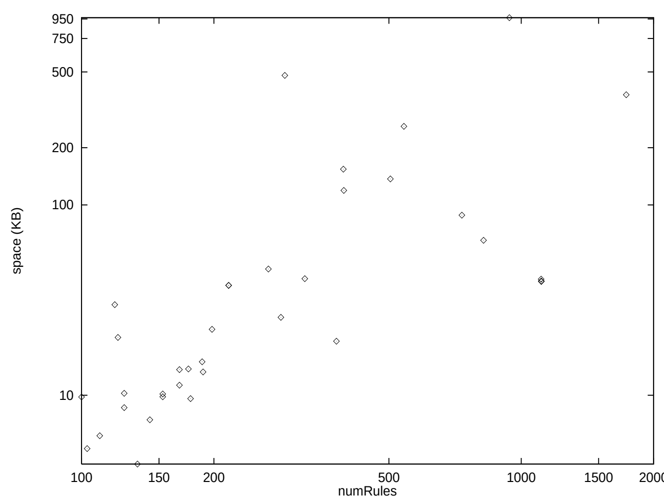


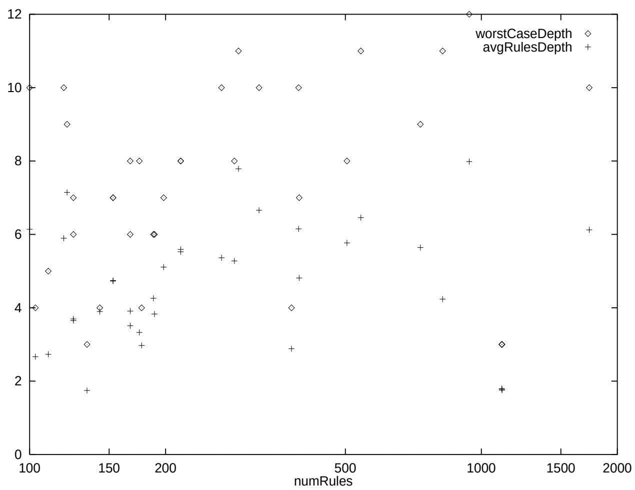
Figure 5 shows the maximum and average tree depth (to calculate the average, we assume that each leaf is accessed in proportion to the number of rules in its *CollidingRuleSet*) for the classifiers with a *binth* of 8 and *spfac* of 4. The worst case tree depth was found to be 12, with an average value close to eight. i.e. in the worst case a total of 12 memory accesses are required, followed by a linear search on eight rules to complete the classification. This makes a total of 20 memory accesses in the worst case.

An important consideration is the preprocessing time

† Grateful acknowledgments to Darren Kerr of Cisco Systems for providing access to these classifiers. These are the same classifiers used in the study described in [2].

required to build the decision tree. This is plotted in Figure 6, showing the highest preprocessing time to be 50.5 seconds, with the next highest taking approximately 20 seconds. All but four classifiers have a preprocessing time of less than eight seconds.

Figure 5: Average and worst case tree depth for binth=8 and spfac=4.



The preprocessing time is clearly quite high, caused mainly by the number and complexity of the heuristics. We expect that this preprocessing time will be acceptable in most applications as long as the time taken to incrementally update the tree is kept small. In practise, the update time depends on the rule to be inserted or deleted. We have found that it takes 1 to 70 milliseconds to incrementally update the data structure on an insertion or deletion of a rule, averaged over all the rules of a classifier. The actual values are plotted in Figure 7.

Figure 6: Time to preprocess the classifier to build the decision tree, measured in seconds. The measurements were taken using the *time()* linux system call in user level 'C' code on a 333MHz Pentium-II PC with 96MB of memory and 512KB of L2 cache.

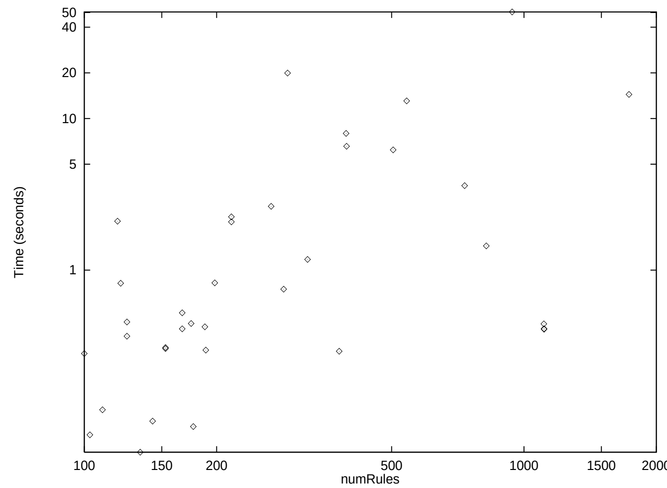
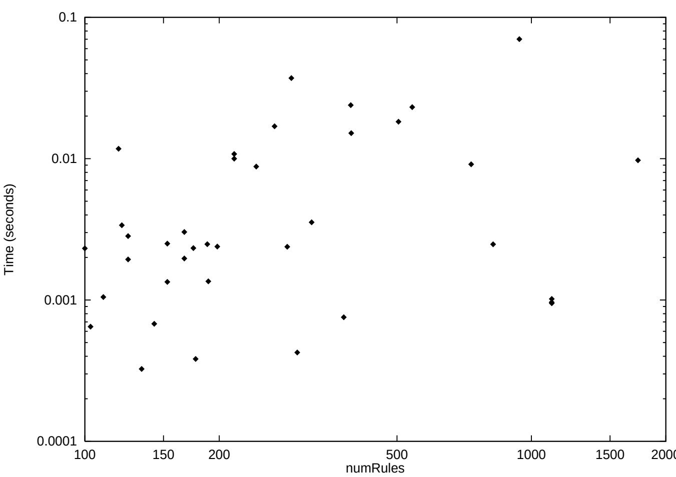


Figure 7: The average update time (over 10000 inserts and deletes of randomly chosen rules for a classifier). The measurements were taken using the *time()* linux system call in user level C code on a 333MHz Pentium-II PC with 96MB of memory and 512KB of L2 cache.



5.3 Variation with *binth* and *spfac*

We show the effect of tuning the parameters, *binth* and *spfac* on the data structure for the largest four dimensional classifier available to us (1733 rules). In our experiments, *binth* takes on the values 6, 8 and 16; and *spfac* takes on the values 1.5, 4 and 8 for each value of *binth*. We make the following observations from our experiments:

1) The *HiCut* tree depth is inversely proportional to *binth* and also to *spfac* (see Figure 8).
2) The data structure storage requirement is directly proportional to *spfac* (as expected) but inversely proportional to *binth* (see Figure 9).
3) The preprocessing time is proportional to the storage consumed by the *HiCut* data structure (see Figure 10).

Figure 8: Showing the variation of tree depth with binth and spfac for a classifier with 1733 rules.

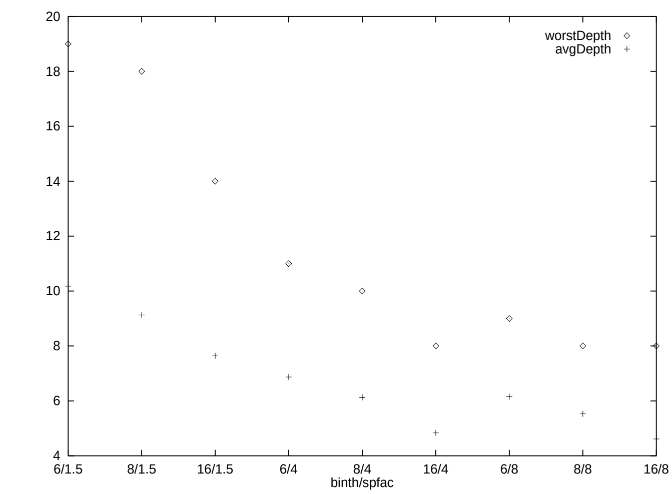


Figure 9: Showing the variation of memory consumption with the binth and spfac parameters.

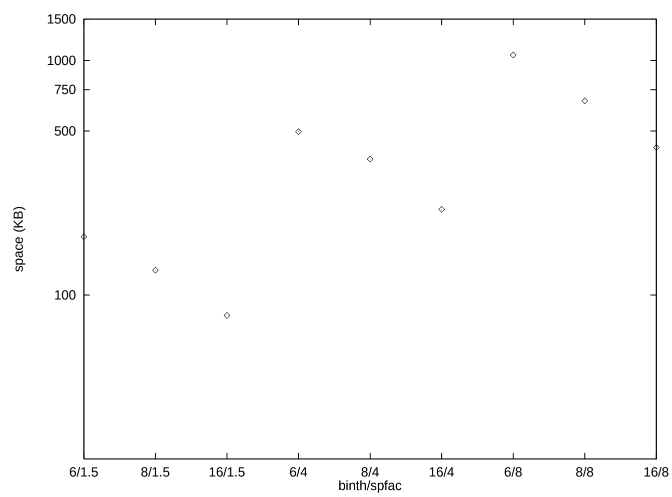
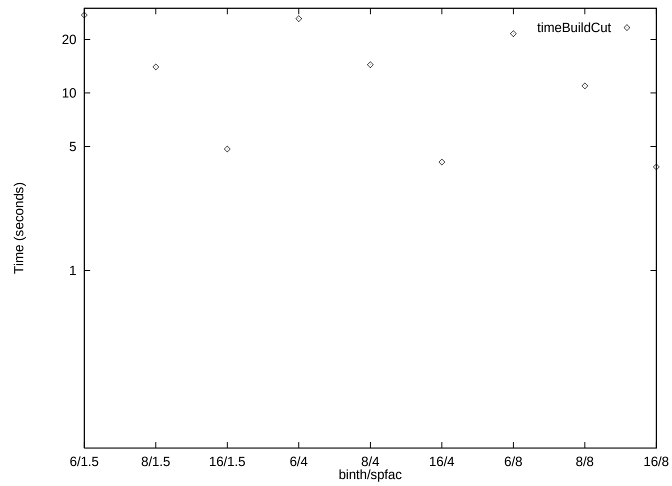


Figure 10: Showing the variation of preprocessing times with binth and spfac.



6 Conclusions

The design of classification algorithms is hampered by worst-case bounds on query time and storage requirements that are so onerous as to make generic algorithms unusable. So instead we must search for characteristics of real classifiers that can be exploited in pursuit of algorithms that are "fast enough" and use "not too much" storage. This task is made harder by the almost complete absence of classifiers (the set available to us is quite small, confidential and from a not particularly diverse range of networks). Even if classifiers were widely available today, it is not clear that they would represent the types of classifiers found in future networks where these algorithms will be used.

But like others before us, we have resorted to heuristics that, while hopefully well-founded in a solid understanding of today's classifiers, make assumptions about the structure of classifiers to reduce query time and storage requirements.

The scheme that we present here, *HiCuts,* performs well on the classifiers available to us, requiring smaller storage and comparable query time when compared with schemes described previously. The complexity of the heuristic means that the decision tree can take tens of seconds to build, but fortunately seems to permit nearly a large number of updates per second.

7 References

[1] A. Brodnik, S. Carlsson, M. Degermark and S. Pink, “Small Forwarding Tables for Fast Routing Lookups.” *Proc. ACM SIGCOMM 1997*, pp. 3-14, Cannes, France.

[2] P. Gupta and N. McKeown, “Packet Classification on Multiple Fields”, *Proc. ACM SIGCOMM 1999*, pp. 147-160

[3] V. Srinivisan, S. Suri and G. Varghese, “Packet Classification using Tuple Space Search”, *Proc. ACM SIGCOMM 1999*, pp 135-146.

[4] M.H. Overmars and A.F. van der Stappen, “Range searching and point location among fat objects”, in Journal of Algorithms, 21(3), pp 629-656, 1996.

[5] T.V. Lakshman and D. Stiliadis, “High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching”, Proc. ACM SIGCOMM, 1998

[6] V.Srinivisan, S.Suri, G.Varghese and M.Waldvogel, “Fast and Scalable Layer4 Switching”, in Proc. ACM SIGCOMM, 1998.

[7] http://www.merit.edu/ipma

8 Appendix

8.1 Algorithm to choose the number of cuts to be made at a node v

/* We are doing a binary search on the number of cuts to be made at this node, v. When the number of cuts are such that the corresponding memory consumption estimate becomes more than what is allowed by the spaceMeasure function *spmf()*, we end the search. It is possible to do smarter variations of this search algorithm.*/

```
n = numRules(v);
nump = max(4, sqrt(n)); /* starting value of number of partitions to make at this node */
for (done=0;done != 0;)
{
 /* assume that the current cut, C, has nump partitions */
 sm(C) = 0;
 for each rule r in CollidingRuleSet(v)
 {
   sm(C) += number of partitions colliding with rule r;
 }
 sm(C) += nump;
 if (sm(C) < spmf(n))
 {
   numP = numP * 2; /* increase the number of partitions */
 }
 else { done = 1;}
}
```

/* we have now found a value of numP (the number of children of this node) which fits our memory requirements criteria */