Detecting and Resolving Packet Filter Conflicts

Adiseshu Hari [1], Subhash Suri[2], Guru Parulkar[2]

[1]Bell Laboratories
101 Crawfords Corner Road
Holmdel, NJ 07733, USA
hari@bell-labs.com

[2]Washington University
Box 1045
St. Louis, MO 63130, USA
(suri, guri)@cs.wustl.edu

Abstract— [1] **Packet filters are rules for classifying packets based on their header fields. Packet classification is essential to routers supporting services such as Quality of Service (QoS), Virtual Private Networks (VPNs), and firewalls. A *filter conflict* occurs when two or more filters overlap, creating an *ambiguity* in packet classification. Current techniques for resolving filter conflicts are based on *prioritizing* conflicting filters, and choosing the higher priority filter. We show that such ordering does not always work. Instead, we propose a new scheme for conflict resolution, which is based on the idea of *adding* resolve filters. Our main results are algorithms for detecting and resolving conflicts in a filter database. We have tried our algorithm on 3 existing firewall databases, and have found conflicts, which are potential security holes, in each of them.**

Keywords—**Packet Filters, Classification, Security, Firewalls**

I. Introduction

The Internet is undergoing fundamental changes both in the demand for bandwidth and in the demand for new services beyond the traditional best effort service, caused by the expanding set of users and the growth of multimedia content. This has fueled the demand for routers able to handle large traffic volumes measuring in millions of packets per second.

At the same time, router technology is advancing from simple destination based forwarding to incorporate a number of new capabilities which affect the forwarding process. Successful deployment of these technologies and services is crucial for the successful evolution of the Internet towards a full service network.

CIDR, IntServ and DiffServ QoS, Firewalls and VPNs are all examples of technologies which have extended the internet forwarding table lookups, from fixed length lookups to sophisticated 5 tuple lookups with wildcarding. What is common in all these examples is that the routers in the networks all have *state* installed in them. The state is an association between a set of packets and the action to be performed on that set. The set of packets is described by the contents of some of the fields in the packets — this is sometimes referred to as a *packet filter*. The packet filters can include fields from the network layer (IP) as well as higher layer (TCP or UDP) fields. For example, an application level flow can be described by a 5-tuple filter consisting of the IP source and destination address, the upper level protocol (TCP or UDP) and the upper level protocol source and destination ports.

In order for the Internet to transform itself from today's chaotic logjam to the full service network of tomorrow, it is important to provide a fast and scalable solution to the problem of packet classification and filter matching. While this problem is under active study these days, a related problem has not received much attention: PACKET FILTERS CAN LEAD TO AMBIGUITIES IN PACKET CLASSIFICATION. This is possible because a packet might match multiple filters, each with a different associated action. This important problem has not been studied so far. We refer to this problem as the problem of *filter conflict and resolution*. This can be split up into two distinct subproblems. First, how does one detect such conflicts? Second, given a set of conflicting filters, how does one resolve these conflicts? This paper presents a general solution to both problems, and a fast solution optimized for commonly occurring filters. It is important to consider filter conflict resolution in any scheme involving filters, since filters, if not handled correctly, can cause packets to be subject to the wrong actions. For example, incorrectly matching packets to filters in firewalls can cause security problems.

CONTRIBUTIONS AND RESULTS.

- We formally characterize the conditions which lead to conflicts amongst filters.
- We prove that existing conflict resolution schemes based on filter ordering do not work in all cases.
- We show that our scheme, based on *adding* new filters, works in all cases.
- Our algorithm is general enough to be applicable for filters with any number of fields.
- When applied to three operational firewall databases, our algorithm uncovered potential security holes, where filter conflicts lead to unintended actions.
- In the special case of 2-tuple filters which correspond to Source-Destination prefix filters, we develop a highly efficient trie-based algorithm which can be used for both filter conflict detection and packet classification, thereby eliminating any redundancy between the control path and the data path.
- We present simulation results showing filter conflict detection times in the order of a few microseconds on a 30000 filter database on a 200 Mhz Pentium workstation.
- We show how to extend the trie-based algorithm to 5-tuple filters, which include protocol and port fields. This extends the applicability of this algorithm to virtually all current uses of filters, ranging from firewalls to VPNs to QoS signaling.

OUTLINE AND ROADMAP OF PAPER

Section II defines filters abstractly and reviews current solutions to the filter conflict problem. We show that the current solutions have potentially undesirable outcomes, caused by implicit and inflexible mapping of packets to filters. This section also

[1]For space considerations all proofs, algorithmic complexity analysis, and some experimental results have been omitted. See [1] for the full length draft

presents a key insight to our solution, namely, a conflict between two filters can be resolved by adding a new filter which covers the region of conflict. We also show why solutions to conflict resolution based on ordering filters do not work in the general case.

Section III provides the algorithm for the general solution of conflict detection in filters with arbitrary number of fields. The remaining sections deal with fast versions of the general algorithm. Section IV provides the motivation for developing such optimized algorithms. Section V provides a solution for 2-tuple filters, for example, filters consisting of IP source and destination address masks. This is important since 2-tuple filters consisting of (IP source, IP destination) addresses are the most widely used filters for unicast and multicast traffic management.

Section VI describes how the 2-tuple algorithm can be extended to the 5-tuple case in which the protocol and port fields are either wildcarded or fully specified. This extends the applicability of our algorithm to virtually all know uses of packet filters.

Section VII describes security holes uncovered by our general algorithm and provides experimental results to verify that the performance of our fast algorithm actually is consistent with the desired goal in terms of time, and stays constant irrespective of the number of filters in the database. Section VIII presents conclusions and plans for future work.

II. Conflicts in Filters

A *filter* F is a k-tuple $(F[1], F[2], \ldots, F[k])$, where each field $F[i]$ is a *prefix* bit string.[2] Each prefix string $x.*$ determines a range of addresses, namely, $[x0\cdots0,\ x1\cdots1]$; the number of bits appended to x is the difference between the maximum bit length of x's field and the number of bits specified in x. For instance, the prefix $10*$ in a 4-bit field determines the range $[1000, 1011] = [8, 11]$. We say that an address X matches a prefix $x.*$ if X lies in the range $[x0\cdots0, x1\cdots1]$. We say that a packet P matches a filter F, if each field of P matches the corresponding prefix of F. For instance, an IP packet with source and destination addresses $(128.112.234.2, 128.122.34.51)$ matches the 2-tuple filter $(128.112.*, 128.122.*)$, but not the filter $(128.112.*, 128.132.*)$.

Each filter F is associated with an action, denoted $A(F)$. A packet P matching a filter F should be processed based on $A(F)$. Typical filters range from 1-tuple to 5-tuple. 1-tuple filters consist of IP destination addresses or prefixes. These are used in routing tables. 2-tuple filters are used in IP multicasting to identify (source, group) pairs and also in VPNs. Here the tuples represent IP source and destination addresses or prefixes. 3-tuple and 5 -tuple filters are used in more refined classification schemes where the protocol and the port fields are also used in acting on a packet. Firewall filters can be range from 1-tuple to 5-tuple and beyond.

The problem with filters is that a packet might match multiple filters with conflicting values for the action. To illustrate the problem of conflicts in filters, let us consider the case of simple 2-tuple filters, consisting of source and destination IPv4 address prefixes. The two fields can either be fully specified or wildcarded, or can be partially wildcarded in standard prefix format. Let $128.112.*$ denote the prefix corresponding to network x and let $128.122.*$ denote the prefix of network y. Consider two filters $F_1 = (128.112.*, *)$, with $A(F_1) = \{100 \text{ Mbps bandwidth}\}$, and $F_2 = (*, 128.122.*)$, with $A(F_2) = \{1 \text{ Mbps bandwidth}\}$. The first filter assigns all packets from source network x a bandwidth of 100 Mbps while the second filter assigns all packets destined to network y a BW of 1 Mbps. What happens if the router starts receiving a 10 Mbps flow from source net x destined to net y? We have a conflict since the packets of the flow match both F_1 and F_2. Which filter should take precedence?

Some possible solutions are:

a) The first matching filter in the filter database takes precedence. For example, if F_1 is stored before F_2 in the database, then the flow goes through at 100 Mbps. On the other hand, if F_2 is stored before F_1, than most of the packets of the flow are dropped, since the flow is restricted to a BW of only 1 Mbps. This approach is commonly used to resolve conflicts in firewalls, where incoming packets are matched against filters specified in access control lists and the action is determined by the first matching filter.

b) Assign priorities to different filters, and use the matching filter with the highest priority. This scheme turns out to be identical to scheme a) if we sort the filters in the order of priority.

c) Assign priorities to fields so that in case of multiple matches the filter with the most specific matching field with the highest priority is selected. For example, if the source address is given higher priority on matches than the destination address, then for packets going from network x to network y the filter F_1 is a better match than F_2.

All of these implicit *conflict resolution schemes*, while simple to implement, suffer from some serious drawbacks. For example, in case a), as we have already discussed, depending on whether F_1 or F_2 is listed, packets of the flow will either go through or be dropped. Thus, this scheme imposes an arbitrariness on the conflict resolution. Case b), as mentioned, suffers from the same problem.

In scheme c), with the filters described above, there is no way to assign a 1 Mbps BW for packets going from net x to net y. Thus, this scheme substitutes arbitrariness with inflexibility in filter matching.

If we substitute a firewall for the QoS aware router described earlier and substitute Accept/Reject for the actions associated with F_1 and F_2, we see that filter conflicts can also lead to security problems. As aptly stated in a book on firewalls —'The point here is that getting filtering rules right is tricky' [2]. In fact, we have actually uncovered similar problems in firewall databases. We formally show in Section II-A why such implicit conflict resolution schemes do not work in the general case.

Our algorithms are based on the following two key observations:

- If filter fields are prefix fields, then each field of a filter is either a strict subset of, or equal to, or a strict superset of, or completely disjoint, from the corresponding field in any other filter. In other words, it is not possible to have partial overlaps of fields. Partial overlaps can only occur when the fields are arbitrary ranges, not

[2] Occasionally a filter field is specified as a range, such as the port field, but we can easily convert an arbitrary range into a small number prefix ranges using well-known techniques.

prefixes.

• By the addition of a new filter (which we call the *resolve filter*) which covers the region of overlap, we can eliminate conflict between two overlapping filters. This assumes a *Best Matching Filter* (BMF) packet classification, analogous to the Best Matching Prefix (BMP) in 1-dimensional lookups. In BMF classification, a packet matches the filter which best matches it in *all* fields. Adding resolve filters guarantees a BMF match.

A. Implicit Conflict Resolution Through Filter Reordering

To see why implicit schemes for conflict resolution based on filter ordering do not work in the general case, consider the example of a company with two geographically dispersed secure divisions. The company policy is to let the secure divisions open TCP connections with the rest of the company, but prohibit the rest of the company from initiating a TCP connection with either division. Assume the IP address prefix of the company is Z, and that of the two divisions are Z.secure1 and Z.secure2. Then an internal firewall inside the company network can contain the following rules:

A From Z.secure1.* to Z.*, 'allow TCP SYN request'
B From Z.secure2.* to Z.*, 'allow TCP SYN request'
C From Z.* to Z.secure1.*, 'reject TCP SYN request'
D From Z.* to Z.secure2.*, 'reject TCP SYN request'

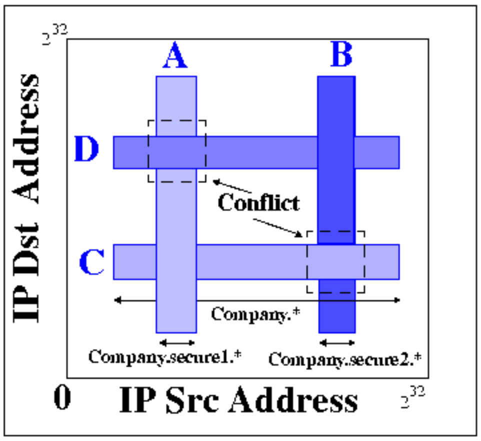

Fig. 1. Conflicting Filters. Scenario where it is not possible to resolve conflicts by reordering filters

The rules are processed based on first matching filter, and are shown graphically in Figure 1. As can be easily seen, there is no explicit rule for the case when nodes in one of the secure divisions attempt to initiate TCP connections with nodes in the other secure division. This corresponds to the regions of overlap of the four filters. In fact, with the current ordering of rules, this case is always allowed. If, however, the company policy is to prohibit TCP connections between the two secure divisions, while allowing TCP connections within the same divisions. it is possible to show that *no permutation of the above four rules can satisfy such a policy [1].* On the other hand, it is always possible to give explicit resolve filters to satisfy such a policy. (Observe that when using our resolve filters, the filter matching is done using the BMF rule.) The resolve filters for our example are:

E From Z.secure1.* to Z.secure2.*, 'deny TCP SYN request'
F From Z.secure2.* to Z.secure1.*, 'deny TCP SYN request'
G From Z.secure1.* to Z.secure1.*, 'accept TCP SYN request'
H From Z.secure2.* to Z.secure2.*, 'accept TCP SYN request'

In the implicit resolution scheme where no resolve filters are added, in order to to obtain the same result as rule E, rule D should have higher priority than rule A. To obtain F, we see that C should have higher priority than B. To obtain G, A should have higher priority than C. From these three rules, we conclude that D should have higher priority than B. However, to obtain H, B should have higher priority than D, which is a contradiction.

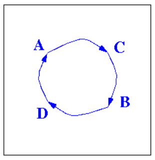

Fig. 2. Transforming conflicting filters into a graph. Cycles indicate conflicts which cannot be resolved by reordering filters

Solving the filter conflict problem by reordering filters can be formulated as a *cycle elimination problem* in a directed graph $\mathcal{D}$. Specifically, we model each filter by a node of a directed graph. We put a directed edge from node F to node G if filters F and G overlap and F has a higher priority than G. For example, the set of filters A, B, C, D described above is transformed into the graph shown in Figure 2. Unambiguous classification by filter ordering is possible if and only if this directed graph $\mathcal{D}$ is *acyclic*. Clearly, if $\mathcal{D}$ contains a cycle, then no reordering of filters can avoid ambiguous classification. But if $\mathcal{D}$ is acyclic, then we can perform a *topological sort* [3] of the nodes to reorder the filters. The following theorem states our result concerning when a set of filters can be made conflict-free with simple filter reordering.

Theorem 1: If graph $\mathcal{D}$ contains a directed cycle, then the set of filters cannot be made conflict free without the addition of a new resolve filter.

We note that a particular cycle can be eliminated by introducing a single resolve filter. For instance, consider the cycle $A \rightarrow C \rightarrow B \rightarrow D \rightarrow A$ of Figure 2. Suppose we add a resolve filter E for the conflicting pair (A, D), and assign E higher priority than both A and D. Thus, in the graph $\mathcal{D}$, the edge $D \rightarrow A$ is replaced by two edges $A \rightarrow E$ and $D \rightarrow E$, and the cycle $A \rightarrow C \rightarrow B \rightarrow D \rightarrow A$ is eliminated. It follows that the smallest number of resolve filters needed to eliminate all conflicts is equal to the smallest number of edges whose removal from $\mathcal{D}$ makes the graph acyclic. Unfortunately, this problem is equivalent to a combinatorial optimization, known as the *smallest feedback arc* problem, which is NP-Complete problem [4]. Thus, *no polynomial-time algorithms exists for determining the smallest number of resolve filter whose addition plus re-ordering will make the filter database conflict-free.*

Our approach in this paper, therefore, is to add resolve filters for each pair of conflicting filters. *Packet classification now corresponds not to the first matching filter, but to the best matching filter.* It is possible to combine our algorithm with some heuristic that tries to break as many cycles as possible with each resolve filter. We have not pursued that strategy yet.

III. The General Algorithm

We recall that each filter F is a k-tuple $(F[1], F[2], \ldots, F[k])$, where each field $F[i]$ is a *prefix* bit string. We say that two prefixes $x.*$ and $y.*$ are *disjoint* if there is no address common to them. Two filters are said to be disjoint if no packet header matches them both.

We start with the most basic question: how to determine if two k-tuple filters have a conflict. The following theorem answers this question by giving a necessary and sufficient condition for detecting the *absence* of a conflict. When the condition fails, we have a filter conflict.

Theorem 2: Two filters F and G are conflict-free if and only if one of the following holds:
1. *There is some field index i such that the prefixes $F[i]$ and $G[i]$ are disjoint, where $1 \leq i \leq k$,*
2. *For all field indices i, $F[i]$ is a prefix of $G[i]$ and a strict prefix for at least one i; or for all indices i, $G[i]$ is a prefix of $F[i]$ and a strict prefix for at least one i.*

Figure 3 presents the pseudo-code for detecting whether two k-tuple filters have a conflict.

```
Algorithm 2FilterConflict (F, G)
      (* Determines if F and G conflict. *)
      for i = 1 to k do
            if F[i] and G[i] are disjoint
                  return "No Conflict";
      Set Flag = 1;
      for i = 1 to k do
            if F[i] is not a prefix of G[i]
                  Flag = 0;
      if (Flag = 1) return "No Conflict";
      Set Flag = 1;
      for i = 1 to k do
            if G[i] is not a prefix of F[i]
                  Flag = 0;
      if (Flag = 1) return "No Conflict";
      else return "Conflict";
end Algorithm
```

Fig. 3. Algorithm to determine if two filters have a conflict.

A. Detecting and Resolving Conflicts

If two filters F, G have a conflict, our solution is to introduce a new filter H, which is the filter corresponding to the overlap region $F \cap G$. We will call H the *resolve filter* for F, G. What are the prefix fields for this resolve filter? It is not difficult to see that each field of H is the longer of the two prefixes in the corresponding fields of F and G. For instance, if $F = (101*, 1*)$ and $G = (10*, 111*)$, then $H = (101*, 111*)$. Figure 4 gives the pseudo-code for computing the resolve filter for two conflicting filters.

```
Algorithm ResolveFilter (F, G)
      (* Computes the filter resolving the conflict
      of F, G. *)
      for i = 1 to k do
            Let x_i be the longer of the two
            prefixes F[i] and G[i];
      return (x_1, x_2, ..., x_k);
end Algorithm
```

Fig. 4. Computing the filter to resolve conflict of two filters.

Figure 5 gives the pseudo-code for detecting and/or resolving conflicts when a new filter is added. The set $C(F)$ in that code stores all the filters in the database B that have a conflict with the newly added filter F. If one is only interested in *detecting* whether a conflict exists, we can quit as soon as Line 4 is executed for the first time. If one is only interested in *enumerating* the filters in conflict with F, we don't execute steps 5–6, and simply list out the set $C(F)$.

Finally, if we want to maintain a conflict-free database, we add the necessary resolve filters whenever a new filter creates conflicts. Suppose that the existing database $B = \{F_1, F_2, \ldots, F_N\}$ is conflict-free. Once the set $C(F)$, containing the filters of B that conflict with F, is determined, we compute the resolve filter corresponding to each F_i in $C(F)$, and add that resolve filter to the database B. A key point to note that is that *adding* a resolve filter *does not require* a recursive call to the algorithm AddNewFilter. This is because adding a resolve filter *does not create* any new conflicts. If the resolve filter were to conflict with any existing filter, it is only because the new filter which introduced the resolve filter in the first place has a conflict with the existing filter. Thus, the worst case running time of the algorithm for adding a filter is $O(N + C)$, where N is the size of the current database and C is the number of conflicts.

```
Algorithm AddNewFilter (F, B)
      (* Insert a new filter into B. *)
1.    Initialize C(F) = {F};
2.    for i = 1 to |B| do
3.          if F and F_i have a conflict then
4.                Add F_i to C(F);
5.    for each filter F' ∈ C(F) do
6.          Add ResolveFilter(F, F') to B;
end Algorithm
```

Fig. 5. Modifying B upon the addition of a new filter.

IV. Fast Filter Conflict Detection

The problem of conflict detection is a problem of the control path, where filters are added. Given a new filter, it is possible to linearly scan the existing set of filters to detect a conflict as described earlier. Is a faster algorithm necessary in the control path? Consider for example, a router processing 100,000 filter updates/s. Note for comparison purposes that large scale telephone switches are designed to handle thousands of calls per second. The router has to process the filter update within 10 microseconds, and that includes the entire control path processing, from receiving the message to admission control, from updating the classifier and the scheduler to propagating any control messages onwards. Clearly, any conflict detection algorithm cannot consume more than a small fraction of the total available time, and therefore has to operate within a few microseconds. The problem becomes more acute when we consider that the filter database can be large — the current Internet backbone routing prefixes themselves number more than 40000. Another point to note about filters is that most filters in the Internet are either 2-tuple filters consisting of IP source and destination address prefixes, or 5-tuple filters which also include the protocol type and upper level port fields. This provides the motivation to develop conflict detection algorithms which are optimized for these cases and whose running time is better than linear. We believe our approach to conflict detection and resolution in filters is fast and scalable enough to be used in the most demanding scenarios of today and beyond. We will cover the performance of our fast algorithm in greater detail in later sections and contrast it with approaches based on a linear search of the database.

V. An Improved Algorithm for 2-Tuple Filters

The key insight for the new algorithm is the following observation, which follows readily from the general result in Theorem 2.

Lemma 3: Filters F and G have a conflict if and only if
1. $G[1]$ *is a prefix of* $F[1]$ *and* $F[2]$ *is a prefix of* $G[2]$, *or*
2. $F[1]$ *is a prefix of* $G[1]$ *and* $G[2]$ *is a prefix of* $F[2]$.

A. Trie Based Conflict Detection Algorithm

We will use the example database B shown in Figure 6 to illustrate our scheme. We develop a *2-dimensional recursive* trie data structure to solve the filter conflict problem. We begin with some basic definitions and facts about tries.

Filter	Source	Destination
F_1	$10*$	$100*$
F_2	$10*$	$011*$
F_3	$10*$	$001*$
F_4	$1*$	$00*$
F_5	$1*$	$11*$
F_6	$1*$	$10*$
F_7	$0*$	$101*$
F_8	$*$	$1*$

Fig. 6. An example 2-tuple filter database.

Recall that a trie is a binary branching tree, with each branch labeled 0 or 1. The bit string associated with a node u is the concatenation of all the bits from the root to the node u. A trie node v is an *ancestor* of another node u if v lies on the path from the root to u. Stated another way, the bit string associated with v is a *prefix* of the bit string at associated with u. If v is a ancestor of u, then u is called a *descendant* of v. (In Figure 7, for instance, the node c is an ancestor of node d and a descendant of node b.)

Our algorithm will need two complementary data structures, one for each of the two cases of Lemma 3. In particular, one data structure can efficiently isolate the filters whose source field is a prefix of F's source field, and then organize these filters to quickly determine if any of them has the destination field with $F[2]$ as a prefix. The second data structure reverses the roles of source and destination fields.

Recursive Trie 1

We start by building a trie on all the source address prefixes in the database B. We call this the *source trie* $S(B)$. Let u be a node in this trie, and let $s(u)$ denote the bit string associated with u. We associate a second trie $D(u)$ (destination trie) with u, which stores the *destination* prefixes of the filters whose source prefix is exactly $s(u)$. More precisely, let us define the set

$$X(u) \;=\; \{F \in B \mid F[1] = s(u)\}.$$

That is, $X(u)$ is the set of filters with source field equal to $s(u)$. We build a secondary trie $D(u)$ on the destination addresses of the set $X(u)$, which is pointed to by the node u. Figure 7 shows the complete construction. Nodes of destination tries are labeled with the filters associated with that destination address.

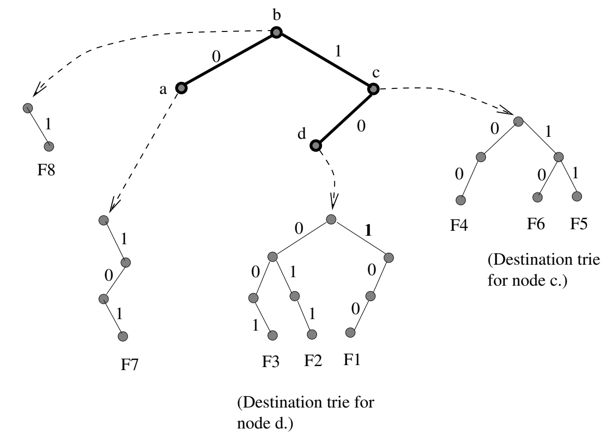

Fig. 7. Recursive Trie 1 for the example database of Figure 6. The source trie $S(B)$ is shown with thick solid lines; various destination tries are shown in thin lines; dashed lines indicate pointers between a node and its associated destination trie.

Recursive Trie 2

The second data structure builds a trie on the destination prefixes in the database B; call this the *destination trie* $D(B)$. For each node u in $D(B)$, let $d(u)$ be the string associated with u. Define the set

$$Y(u) \;=\; \{F \in B \mid F[2] = d(u)\}.$$

$Y(u)$ is the set of filters with destination field equal to $d(u)$. We build a secondary trie $S(u)$ on the source addresses of the set $Y(u)$, which is pointed to by the node $u \in D(B)$.

B. The Conflict Detection Algorithm

Suppose we want to add a new filter F to an existing database B and, if there are conflicts, modify the database by adding appropriate resolve filters. We will search both Recursive Tries 1 and 2. When searching Recursive Trie 1, we first use the source trie $S(B)$ to locate the *longest matching prefix* of the source field $F[1]$. Let u be the node with this longest matching prefix, and let $v_1, v_2, \ldots, v_m$ be the nodes in the source trie whose bit strings correspond to (proper) prefixes of $F[1]$. Observe that $v_1, \ldots, v_m$

are all ancestors of u, and possibly $u = v_1$. We visit each of the destination tries $D(v_1), D(v_2), \ldots, D(v_m)$ in turn. In each destination trie, say, $D(v_i)$, we locate the longest matching prefix of the destination field $F[2]$. If the node z with the longest matching prefix is a *leaf* then no filters in $D(v_i)$ are in conflict with F. Otherwise (if z is not a leaf), *all* the filters associated with *descendants* of z are in conflict with F.

The search in the second structure, Recursive Trie 2, is similar, except the roles of source and destination fields are reversed. The completes algorithm is given in Section V-C.

As an example, suppose we want to add a filter $F = (10*, 1*)$ to the database of Figure 6. Then, the node d of the source trie gives us the longest matching prefix for the source field $10*$. The ancestors of d associated with prefixes of the source string $10*$ are b and c. We search the destination trie pointed to by c to locate the longest matching prefix of the destination field $1*$. The node with this string is *not* a leaf, and so all the filters that descend from it are in conflict with F. There are two such filters, F_5 and F_6. Indeed, it is easy to see that $F_5 = (1*, 11*)$ and $F_6 = (1*, 10*)$ both have a conflict with $F = (10*, 1*)$. Next, we search the destination trie pointed to by b, and find that the node with destination prefix $1*$ is a leaf. A search of Recursive Trie 2 discovers no new filter conflicts, and so there are only two conflicts.

C. Pseudocode of conflict detection algorithm for 2-tuple filters.

Algorithm FastDetect (F, B)
1. Initialize $C(F) = \{F\}$;
   (* Search Recursive Trie 1 *)
2. Let u be the node in the source trie $S(B)$ for which $s(u)$ is the longest matching prefix of the source field $F[1]$;
3. Let $v_1, v_2, \ldots, v_m$ denote the nodes in $S(B)$ whose bit strings correspond to (proper) prefixes of $F[1]$;
4. **for** $i = 1$ to m **do**
5. Determine the node z in $D(v_i)$ whose string is the longest matching prefix of the destination field $F[2]$;
6. Add to $C(F)$ all the filters stored with a descendant node of z;
   (* Search Recursive Trie 2 *)
7. Let u be the node in destination trie $D(B)$ for which $d(u)$ is the longest matching prefix of the destination field $F[2]$;
8. Let $v_1, v_2, \ldots, v_m$ denote the nodes in $D(B)$ whose bit strings correspond to (proper) prefixes of $F[2]$;
9. **for** $i = 1$ to m **do**
10. Determine the node z in $S(v_i)$ whose string is the longest matching prefix of the source field $F[1]$;
11. Add to $C(F)$ all the filters stored with a descendant node of z;
12. If $C(F)$ only contains F, then add F to B, and **return** "No Conflict";
13. **for** each filter $F' \in C(F)$ **do**
14. Add ResolveFilter(F, F') to B;

end Algorithm

D. Improving Conflict Detection Times

The straightforward trie based search outlined above has the advantage of an update time independent of the number of filters in the database. However the lookup times are of $O(w^2)$, where w is the width of each field. This is because when a lookup in the second level trie fails, we backtrack and restart at the root of the next first level trie. It is possible to use *precomputation* and *switch pointers* [5] to speed up search in a later trie based on a search in an earlier trie. We do not present the details here, since the addition of switch pointers and the necessary precomputation is already explained elsewhere [5]. The main point to note is that it is possible to reduce the conflict detection time from $O(w^2)$ to $O(w)$ (More precisely $O(\log(w) + w/k)$ for a k-bit expanded trie). The tradeoff is that the precomputation involved raises the filter update time to $O(N)$.

VI. EXTENDING FASTDETECT TO 5 TUPLES

We have seen how FastDetect can be used to speed up the search for conflicting filters in a database consisting of 2-tuple filters, with each tuple containing prefixes. Such 2-tuple filters can be used to represent host to host or network to network or IP multicast flows. However, 2-tuple filters are not capable of representing application to application flows, or even host to host flows with greater granularity. As we discuss later, it is difficult to extend FastDetect to prefixes in more dimensions without exceeding both computational and memory limits. However, there are a number of special cases of 5-tuple filters in which it is possible to use FastDetect. This is because the other fields of interest in IP packet classification, namely the protocol type and the source and destination ports, are usually either fully wildcarded, or fully specified.

A. Extending FastDetect to 3 Tuples

We first begin by extending FastDetect to support the protocol field. The key idea here is to partition the set of 3-tuple filters into disjoint sets of 2-tuple filters. Clearly, if we have multiple disjoint sets of 2-tuple filters and we wish to see if a new filter conflicts with any of the existing filters, it is sufficient to separately check the new filter with each set of disjoint filters using FastDetect.

Consider 3-tuple filters consisting of IP source and destination address prefixes and and the protocol type. We restrict the protocol field to be either TCP, UDP, or wildcarded. In case the protocol field is wildcarded, we replicate the filter three times, once with the protocol field set to TCP, once to UDP and once to the special value OTHER. This approach is similar to the one taken in [5]. Thus, at the expense of a possible three fold increase in memory, we can partition the set of filters into three disjoint sets. Clearly, there is no overlap or conflict between the three sets. Given a new 3-tuple filter, we can see which set of filters to check for conflicts, based on the protocol type specified in the filter. For example, if it is a TCP filter, it will not conflict with either the set of UDP filters or the set of OTHER filters. Thus we only need to check for conflicts with the set of TCP filters. In this case, the time taken for conflict detection, as well as the FastDetect algorithm remains unchanged. The only overhead is the three fold increase in memory for filters with wildcarded protocol field.

B. Proceeding to 5 Tuples

We restrict the source and destination ports to be either fully wildcarded or fully specified. Now, we need to store multiple filters at each leaf node, since in the 5-tuple case, multiple filters could have the same source and destination address prefix and protocol type. These filters differ only in their source and destination ports. We divide these filters into four sets and have pointers pointing to each set from the node in the trie corresponding to the source and destination prefixes of the filter in the sets. The first set consists of both source and destination ports wildcarded. Clearly, there can only a single such entry for each node. The second set consists of those filters for which the source port is well specified and the the destination port is wildcarded. The third set consists of those filters whose source port is wildcarded and the destination port is well specified. The fourth set consists of filters with well specified source and destination ports. We refer to these four sets as the $(*,*)$, $(s,*)$, $(*,d)$ and the (s,d) sets respectively.

We run the same FastDetect algorithm with some modifications on the two recursive tries to be searched. We present the modification for the trie with source addresses at the top level and destination addresses at the second level. The modification is identical for the search on the other recursive trie. We maintain two additional variables, which report the nature of the prefix match in each of the two levels. The first variable, called *SrcPrefixLen* is set to *longer* or *equal* depending on whether the source address prefix of the new filter is longer or equal to than the current entry in the trie it is being compared to. Note that the nature of the algorithm is that we do not proceed with the second level search when the source prefix of the new filter is shorter than the entries in the first level trie. Therefore we do not now consider the case when *SrcPrefixLen* set to *shorter*, though we get back to this case later. At the second level trie, at each step, we set *DstPrefixLen* to be either *longer*, or *equal* or *shorter* depending on whether the destination prefix of the filter is longer, equal to or shorter than the destination prefixes of the filters stored at the node on that step. Recall that at each node, the filters are stored in the form of four sets: $(*,*)$, $(s,*)$, $(*,d)$ and (s,d) based on whether the port fields of the filters are well specified or wildcarded. What we are interested in is the following: do the four sets of filters stored at the node conflict with the new filter, given the new filter and the current value of *SrcPrefixLen* and *DstPrefixLen*? Table I gives us the answer for the case when the the input filter has the destination port wildcarded. The table shows the collection of conflicting filters for different combinations of port fields in the newly input filter and existing filters. The first column shows the possible values of the port fields of the new filter and the values of the two variables at a particular step along the second trie. The other columns show the possible conflicts with filters in each of the four sets of filters stored at that node. We can construct similar tables for other cases of the input filter, for example, when the input filter has both ports wildcarded, or both ports well specified [1]. For example, consider a new filter (IP Src, IP Dst, TCP, s_i, *), in which the source port is well specified and the destination ports is wildcarded. We begin by traversing the tries containing TCP filters. Remember that there are two such tries, one with the source address on the first level and the other with the destination address on the first level. Assume we are traversing the trie with the source address on the first level. At a given step on the second level, assume *SrcPrefixLen* is set to *longer* and the *DstPrefixLen* is set to *shorter*. What that means is that the source address prefix of the filter is longer and the destination address prefix shorter than any filter stored at that node. In the 2-tuple case, this would automatically cause a conflict. However, in the 5 tuple case, this can cause a conflict only if the source and destination ports and the protocol fields overlap too. The protocol field is already the same, because of the way the filters are partitioned. Thus we need to check if the source and destination ports overlap with the existing filter. As can be easily seen, they overlap when either the stored filter has both ports wildcarded, or the source port wildcarded and the destination port well specified, , or the source port the same as the source port of the new filter and the destination port well specified, We get exactly this information from Table I, namely, that in such a case we conflict with the filter stored in the set $(*,*)$, conflict with the filter in the $(,*)$ set with source port equal to s_i, conflict with all filters in the $(*,d_i)$ set and conflict with the filter in the set (s,d) which has source port s_i. What Table I tells us is that at each step along the second trie, we only have to consider a subset of the 4 set of filters stored at that particular step and within that subset we only have to consider a subset of the filters in that set.

While Table I lists 6 different combinations of the two variables, we note that the case when *SrcPrefixLen* is *longer* and *DstPrefixLen* is *equal* is identical to the case when *SrcPrefixLen* is *equal* and *DstPrefixLen* is *longer*. As a result we can club these together into one case, reducing the total number of cases.

One case that we have not covered is the case when both source prefixes of the new filter are shorter than the existing filters in the database. To cover this case, if during the search of the source addresses on the first level trie, we see that *SrcPrefixLen* is going to be set to the value *shorter* we make a note of the current node in the trie and proceed to the other 2 level trie. If during the search of the destination addresses on this trie, we see that *DstPrefixLen* is going to be set to *shorter*, then we mark this node too. we now have to do an exhaustive search down one of the two marked nodes. If each node were to carry a count of descendant filters, then one obvious optimization is to traverse that node which has the least number of descendants. We use a table similar to Table I to determine conflicts in this case. One optimization in this case is that we do not need to search for conflicts if the new filter has both ports wildcarded.

VII. Experimental Results

A. Firewall Conflict Detection

We used our general algorithm on 3 sets of firewall databases from existing commercial organizations. In all three cases we were able to uncover filter conflicts which could allow malicious users unintended access to internal company sites. While we are unable to provide the exact details of the problem due to the confidential nature of the firewall databases, we discuss the problem in general terms. In firewalls, the rules are processed sequentially, with the first matching rule being applied to the packet. In case of all three firewall databases, unrestricted access was

TABLE I
CONFLICT DETECTION WHEN INPUT FILTER HAS DESTINATION PORT WILDCARDED

New Filter Ports, *SrcPrefixLen*, *DstPrefixLen*	$(*,*)$	$(s,*)$	$(*,d)$	(s,d)
$(s_i,*)$, *longer*, *longer*	No Conflict	No Conflict	Conflict	Conflict with (s_i,d)
$(s_i,*)$, *longer*, *equal*	No Conflict	No Conflict	Conflict	Conflict with (s_i,d)
$(s_i,*)$, *longer*, *shorter*	Conflict	Conflict with $(s_i,*)$	Conflict	Conflict with (s_i,d)
$(s_i,*)$, *equal*, *longer*	No Conflict	No Conflict	Conflict	Conflict with (s_i,d)
$(s_i,*)$, *equal*, *equal*	No Conflict	No Conflict	Conflict	No Conflict
$(s_i,*)$, *equal*, *shorter*	Conflict	No Conflict	Conflict	No Conflict

granted to external packets sourced *from* a given port x. This was followed later down the database by other rules rejecting external packets *destined to* a set of ports Y. As can be seen, there exists a *conflict now for external packets sourced from port x and destined to any of the set of ports in Y*. In the current configuration of the firewall database, such packets can get through, which is not the intended action.

B. Implementation Details

Due to patenting issues we did not implement the switch pointer based algorithm described above. We implemented the FastDetect algorithm using a 1 bit trie. The aim of the implementation was to verify experimentally the complexity of the algorithm and the running time for different sizes of filter databases and to obtain insight into the nature of conflicts for filters of varying degrees of wildcarding. The implementation consists of two procedures: an *insert()* procedure to insert a filter into the database of filters, and a *detect()* procedure to detect if a new filter conflicts with the database (the actual FastDetect algorithm). The *detect()* procedure works in one of two modes. In the *quick* mode, it reports whether the new filter conflicts with the existing set of filters in the database or not. In the *detailed* mode, it not only reports whether there is a conflict or not, but also enumerates the list of conflicting filters.

In the absence of any publicly available filter database, we constructed an artificial database by generating IP source and destination addresses and masks using a uniform random number generator. The length of the source and destination masks were uniformly varied from 5 bits to 9 bits to simulate a range of address masks. The implementation was a user level program on the NetBSD 1.3 operating system running on a 200 Mhz Pentium Pro workstation. We now discuss the individual results.

C. Conflict Detection

Figure 8 plots the time taken for the FastDetect algorithm in the *quick* mode to detect the presence of a conflict when a new filter is added to an existing set of filters. The number of filters in the database was varied from 10 to 30000. For a given size of the database, the figure shows the average time taken for conflict detection for a new filter. The average time was computed by running the algorithm 1000 times, each time with a randomly generated filter, and then averaging the total time. Two curves are shown for the conflict detection times—one for the cached case and the other for the non-cached case. In the cached case, the algorithm was run repeatedly 1000 times with the same filter, while in the non-cached case the algorithm was run only once with each filter. As expected, the cached times are much better than the non-cached times. Also, as can be seen, the conflict detection time rises much more slowly than the size of the database. As the size of the database increases from 10 to 10000, the conflict detection time rises from around 2 microseconds to 6 microseconds and levels off there. Essentially, once the database size exceeds a certain limit, the algorithm takes a constant amount of time to detect conflicts, matching the theoretical bound $O(w^2)$, which is independent of the database size. We noted in the introduction that with a target signaling rate of 1000 filter updates/s, the time to detect conflicts should be be a small fraction of 1000 microseconds, which is met by our implementation.

Figure 8 also plots the conflict detection time using a linear search of the database. Our efficient algorithm FastDetect outperforms the linear search even for database size as small as 20-30. For databases of 3000 filters, the linear search is full *two orders of magnitude* slower than FastDetect. We also observe that there is little difference between the cached and the uncached cases for the linear search, since the linear search through the database is not able to make effective use of the data cache.

D. Number of Conflicts per Filter

Next, we now look at the number of conflicts created by adding a new filter to an existing set of filters. Figure 9 shows the histogram of conflicts created by the addition of a typical filter to a database with 1000 filters, and a database with 10,000 filters. We ran FastDetect in *detailed* mode 1000 times, adding a new filter each time, and plotted the number of conflicts for each of those 1000 runs. As expected, for the 10K filter database, adding a new filter causes an order of magnitude more conflicts than adding a filter to the 1K filter database.

We also experimented with the distribution of filters masks. In the histogram of Figure 9, each filter field is at least 4 bits long, and we let masks vary uniformly between 5 and 9 bits. As seen

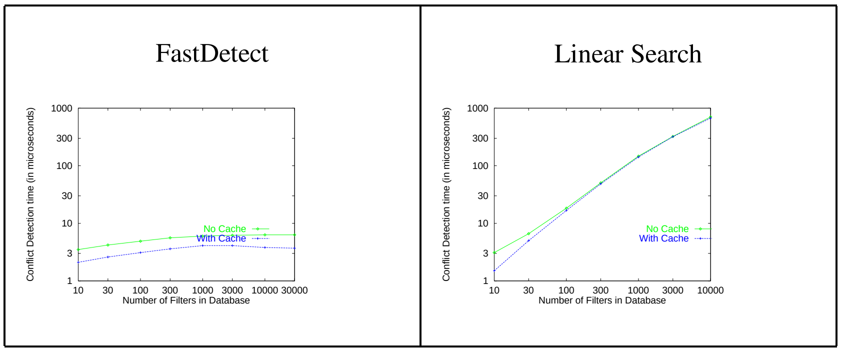

Fig. 8. Conflict detection time. fast detect is orders of magnitude faster than a linear search once the database size rises beyond 1000 filters

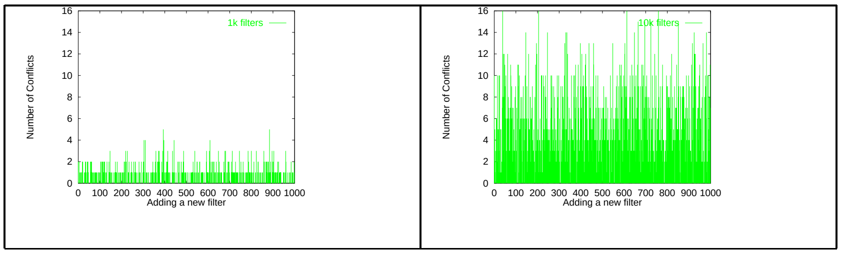

Fig. 9. Typical number of conflicts for each filter. Mask varies from 5 bits to 9 bits

in the figure, the number of conflicts is relatively small, always less than 16. If we require longer masks, such as between 8 and 15 bits, the number of conflicts decrease even further. In fact, the number of observed conflicts almost dropped to zero. In general, the better specified a filter (the longer the mask), the less likely it is to cause a conflict. Conversely, the less specified a filter (fewer mask bits), the more likely it is to generate conflicts.

E. Conflict Enumeration

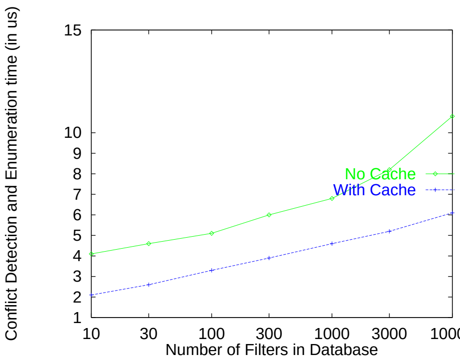

Fig. 10. Conflict enumeration using Fast Detect.

How much time does FastDetect take to enumerate the list of conflicting filters, as opposed to simply detecting the presence of a conflict? Of course, this depends on the number of conflicts.

Figure 10 shows the time taken for the FastDetect algorithm in the *detailed* mode to detect conflicts and to list the set of conflicting filters. This is with a filter database with address masks between 5 bits and 9 bits. In comparison with Figure 8, we see that for small databases, the time taken for *quick* mode and *detailed* mode are roughly the same, but as the size of the database increases, the time taken for enumerating the conflicting filters dwarfs the time taken for simple conflict detection. This again matches the theoretical observation that the conflict enumeration time is linear in the number of filters in the database. However, it is important to note that the time constant is heavily dependent on the filter distribution. In case the database contains lots of less specified filters with fewer mask bits, there are likely to be a lot more conflicts, and correspondingly, the time taken to enumerate the conflicts will increase.

VIII. Conclusions

Filters are powerful tools for reducing state in networks supporting QoS. However, multiple overlapping filters can cause conflicts in mapping packets to filters. This paper describes the problem and presents innovative solutions based on a geometric transformation of the problem.

A general solution is presented for the k-tuple filter, and an optimized version is described for the more common 2-tuple filters consisting of source and destination addresses. We also show how to use the 2-tuple algorithm for the 5-tuple case in which the other three tuples have a restricted set of values. The 2-tuple algorithm is able to detect conflicts in a time independent of the number of filters in the filter database, and is able to resolve them in a time linear with the number of conflicting filters. We have experimentally validated these results using a 1-bit trie implementation, which yielded conflict detection and resolution time in the order of microseconds on a 200 Mhz Pentium workstation. We forsee three principal uses of this algorithm. One is to analyze existing filter databases in firewalls and QoS aware routers to detect conflicts. We have already detected conflicts in firewall databases from one organization and are currently looking at databases from a different organization. Another use is in DiffServ networks at network entry points where the border routers will mark specific packets with flow labels based on state setup in the filter database. The third is in next generation signaling protocols which will carry filters and related packet handling information [6], [7]. As the signaling information propagates through the network, network routers can use the algorithm to report conflicts back to the originators of the signaling requests. It could be argued that since resolving conflicts by adding new resolve filters would require policy input and possibly human input, there is no need for a fast conflict detection and resolution algorithm. However, as these examples illustrate, the conflict detection is done at one site (a router processing the signaling message) and the conflict resolution is done elsewhere (the source of the message). It is important for routers to be able to process signaling messages as fast as possible in order to leave enough processing power for other tasks like packet forwarding, scheduling, routing updates and other signaling requests. Besides these three applications, the general notion of conflict detection and resolution can be applied in a variety of scenarios ranging from digit analysis in telephone networks to policy consistency checking in organizational policies.

Related Work

The problem of conflict detection amongst IP filters has been described in [8]. They describe a similar approach to resolving conflicts by explicitly adding new filters. However, they they do not describe any algorithm for detecting filter conflicts.

Conflict detection among filters is related to the abstract problem of *multidimensional range searching*, which is studied in computational geometry. The filter database corresponds to a set of N k-dimensional rectangular boxes, and the object is to determine the subset of boxes that intersect a query box. Edelsbrunner [9] has proposed a data structure that can solve this problem in worst-case time $O((\log N)^{2k-1} + R)$, where R is the number of rectangular boxes intersecting the query box. The data structure requires $O(N(\log N)^{k-1})$ space and $O(N(\log N)^{k-1})$ construction time.

Unfortunately, the filter conflict problem is somewhat different from the multi-dimensional rectangle intersection problem—while a filter contained inside another filter (a filter that is more specific than the other in all fields) is not a conflict, the corresponding rectangles are considered intersecting in the geometric framework. Thus, the number of rectangle intersections R can be much bigger than the number of filter conflicts C. Secondly, even for modest values of N and k, the worst-case time and space bound guaranteed by this data structure are hopelessly bad. For instance, when $N = 10,000$ and $k = 4$, the algorithm guarantees a worst-case search cost of $13^7 = 62748517$, meaning that it is no better than a linear search through the filters.

Recently, the compaction problem for filter databases has been studied in [10]. In particular, given a filter database D, they construct a new database with fewest number of filters which has the same classification behavior as D. Interesting, the algorithm works only for conflict-free 2-tuple filters.

Conflict-free filters have another unexpected side effect. A lower bound in [11] shows that for arbitrary 2-tuple filters, $2w-1$ memory accesses are needed in the worst case to classify a packet with w bits per field. However, recently, [12] shows an $O((\log w)^2)$ algorithm for packet classification in 2-tuple filters if the filters are conflict free. Thus, in addition to specifying unambiguous classification policy, conflict free filters also yield improved classification algorithms.

Implementation Optimizations and Packet Classification

Fast multibit hashed trie lookups designed for IP prefix matching [13] can reduce the conflict detection times to nanoseconds. The advantage of the current scheme is that the *insert()* procedure is straightforward, while the fast schemes have slower *insert()* procedures, besides increased memory cost.

As described, the algorithm and trie data structures are used only for filter conflict detection in the control path. It is however, possible to reuse the tries for packet classification in the data path as well. The main issue here is the speed. Using the trie for packet classification yields lookup times comparable to the time taken for conflict detection, in the order of a few microseconds with our current 1 bit implementation. As explained above, it is possible to reduce this by a factor of k while increasing memory utilization 2^k times by using k-bit tries. 4 bit tries, for example, can reduce average lookup times to around 1 microsecond. While this is not state of the art, this offers a good blend of memory utilization and fast insert times, which makes it suitable for edge routers which do not have a very high packet throughput, but which need to handle a lot of signaling messages.

Handling Ranges

Both the general algorithm and the optimized version for the 2-tuple case work on filters with prefix strings. However, filters are sometimes specified as ranges. For example, a firewall might block an arbitrary range of ports. It is always possible to transform a subrange of [0, 2^k] into at most $2k$ prefixes [5]. Thus, a filter with arbitrary ranges can always be transformed into a set of filters with prefix strings. We therefore do not view this as a limitation of our algorithm.

References

[1] Hari Adiseshu, Subhash Suri, Guru Parulkar, "Packet Filter Management for Layer 4 Switching—Unpublished Draft," in *http://www.ccrc.wustl.edu/˜hari/packet-filter.ps* , 1999.

[2] Brent Chapman and Elizabeth Zwicky, *Building Internet Firewalls*, O'Reilly Associates.

[3] Thomas H Cormen, Charles E Lieserson and Ronald L Rivest, *Introduction to Algorithms*, The McGraw-Hill Book Company, 1990.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

[5] V. Srinivasan, G. Varghese, S. Suri and M. Waldvogel, "Fast and Scalable Layer Four Switching," in *Sigcomm*, 1998.

[6] B. Braden, L. Zhang, S. Berson, S Herxog and S Jamin, "Resource Reservation Protocol (RSVP) – Version 1 Functional Specification," in *RFC2205*, September 1997.

[7] Hari Adiseshu, Guru Parulkar and Raj Yavatkar, "A State Management Protocol for IntServ, DiffServ and Label Switching," in *IEEE ICNP*, 1998.

[8] D. Decasper, Z. Dittia, G. Parulkar, B. Plattner, "Router Plugins: A Modular and Extensible Software Framework for Modern High Performance Integrated Services Routers," Washington university technical report: Wucs-98-08, February 1998.

[9] H. Edelsbrunner, "A New Approach to Rectangle Intersections (Parts I and II)," in *Int. J. of Computer Math.*, 1983, pp. 209–229.

[10] Subhash Suri, Thomas Sandholm, Priyank Warkhede, "Compaction of Routing Tables and Packet Filters," in *Submitted for Publication*, 1999.

[11] V. Srinivasan, G. Varghese, S. Suri, "Packet Classification using Tuple Based Search," in *Sigcomm*, 1999.

[12] Priyank Warkhede, Subhash Suri, George Varghese, "Fast Classification in Conflict Free Filters," in *Submitted for Publication*, 1999.

[13] Waldvogel et. al., "Scalable High Speed IP Routing Lookups ," in *Computer Communication Review, Vol 27, #4*, October 1997.