Conflict Detection And Resolution In Two-dimensional Prefix Router-Tables

Haibin Lu & Sartaj Sahni
{halu, sahni}@cise.ufl.edu
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611

Abstract

We show that determining the minimum number of resolve filters that need to be added to a set of 2-dimensional prefix filters so that the filter set can implement a given policy using the first-matching-rule-in-table tie breaker is NP-hard. Additionally, we develop a fast $O(n \log n + s)$ time, where n is the number of filters and s is the number of conflicts, plane-sweep algorithm to detect and report all pairs of conflicting 2-dimensional prefix filters. The space complexity of our algorithm is $O(n)$. On our test set of 15 2-dimensional filter sets, our algorithm runs between 4 and 17 times as fast as the 2-dimensional trie algorithm of Hari et al. [9] and uses between 1/4th and 1/8th the memory used by the algorithm of [9]. On the same test set, our algorithm is between 4 and 27 times as fast as the bit-vector algorithm of Baboescu and Varghese [2] and uses between 1/205 and 1/6 as much memory. We introduce the notion of an essential resolve filter and develop an efficient algorithm to determine the essential resolve filters of a prefix filter set.

Keywords: Packet classification, 2-dimensional prefix filters, filter conflict.

1 Introduction

An Internet router classifies incoming packets into flows[1] utilizing information contained in packet headers and a table of (classification) rules. This table is called the **router table** (equivalently, **rule table**). Each router table rule is a pair of the form (f, a), where f is a filter and a is an action.

[1]A **flow** is a set of packets that are to be treated similarly for routing purposes.

The action component of a rule specifies what is to be done when a packet that satisfies the rule filter is received. Sample actions are drop the packet, forward the packet along a certain output link, and reserve a specified amount of bandwidth. The filter component of a rule is a k-tuple the fields of which may represent, for example, the source address of the packet, the destination address, protocol, and port number. Each field of a k-tuple may be specified as a single value, a range or a prefix. A destination address field that is specified as a range $[u, v]$ **matches** the destination address d iff $u \leq d \leq v$, while a destination address field specified by the prefix r matches all destination addresses that begin with r[2]. A filter f matches a packet p iff every field of f matches the corresponding value of p (i.e., the destination field (if any) of f matches the destination address of p, the source address field (if any) of f matches the source address of p, the port number field (if any) of f matches the port number of p, etc.). We may assume that no two rules of the router table have the same filter.

Since an Internet router table may contain several rules that match a given packet p, a tie breaker is used to select a rule from the set of rules that match p. Some commonly used tie breakers are (a) select the first rule in the table that matches p, (b) select the highest-priority rule that matches p, and (c) select the most-specific rule that matches p[3].

In the **packet classification** problem, we wish to determine which rule of the router table is to be applied to a given packet[4] Data structures to represent one-dimensional router tables (i.e., tables in which every filter has a single field, which is typically the destination address of the packet being classified), have been extensively studied. These structures are reviewed in [13] and

[2]For example, the prefix 10* matches all destination addresses that begin with the bit sequence 10; the length of this prefix is 2.

[3]Let f and g be two filters. f is more specific than g iff every packet matched by f also is matched by g and there is at least one packet matched by g that is not matched by f.

[4]In some packet classification applications it is desirable to report all rules that match a given packet. In this paper, however, our focus is applications in which only one matching rule is to be reported. A tie-breaker is provided to uniquely determine which of a set of matching rules is to be reported.

[14], for example. Although 1-dimensional prefix filters are adequate for destination based packet forwarding, higher dimensional filters are required for firewall, quality of service, and virtual private network applications, for example. Two-dimensional prefix filters, for example, may be used "to represent host to host or network to network or IP multicast flows" [9] and higher dimensional filters are required if these flows are to be represented "with greater granularity." Eppstein and Muthukrishnan [6] state that "Some proposals are underway to specify many fields ... while others are underway which seem to preclude using more than just the source and destination IP addresses ... (in IPsec for example, the source or destination port numbers may not be revealed)." Kaufman et al. [18] also point out that in IPsec, for security reasons, fields other than the source and destination address may not be available to a classifier. *Thus two-dimensional prefix filters represent an important special case of multi-dimensional packet classification.* Data structures for multi-dimensional (i.e., $k > 1$) packet classification are developed in [1, 2, 3, 5, 6, 7, 8, 9, 11, 12, 15, 16], for example. In the sequel we use the terms rule and filter interchangeably because the filters in a rule-table are distinct and, in this paper, we are not concerned with the action associated with a rule.

Hari et al. [9] introduced the notion of **filter conflict**. Two filters f and g are in **conflict** iff the following three conditions hold:

1. There is at least one packet that is matched by both f and g.

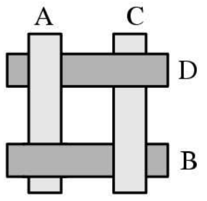
2. There is at least one packet that is matched by f but not by g.

3. There is at least one packet that is matched by g but not by f.

Note that when our filters are one-dimensional prefix filters, it isn't possible to have filter conflict. Figure 1(a) shows an example router-table with 4 2-dimensional prefix filters A, B, C, and D. These filters are shown as rectangles in Figure 1(b). Each rectangle represents the set of filters

Filter	Source Prefix	Destination Prefix
A	001*	*
B	*	001*
C	110*	*
D	*	110*

(a) 4 Filters



(b) Rectangular representation

Figure 1: Conflicting filters

it matches. Conflicts exist between the pairs (A, D), (D, C), (C, B) and (B, A), and the conflicts are represented by the overlap region between pairs of rectangles. The overlap between rectangles A and D, for example, represents packets matched by both A and D. The example of Figure 1 is essentially the same as that given in [9]. Since neither A nor B is more specific than the other, the most-specific-rule tie breaker cannot be used to decide how to classify packets that lie in the overlap region of A and D. Now suppose that the desired classification policy is that filter A is to be selected for packets in the overlap region between A and D, D is selected between D and C, C is selected between C and B, and B is selected between B and A. If we use the first-matching-rule-in-table tie breaker, then A must precede D in the table, D must precede C, C must precede B, and B must precede A. This set of precedence requirements is cyclic and so is impossible to accomplish. The same cyclic requirement arises if we attempt to use the highest-priority tie breaker. So none of the stated tie breaker schemes may be used to achieve the desired classification policy.

To get around this difficulty, Hari et al. [9] propose the introduction of resolve filters. A **resolve filter** is a filter that matches the packets in the overlap region of two conflicting filters. Let E be the resolve filter for A and D, and let the action associated with E be the same as that associated with A. We can get the classification policy stated above by ordering the filters as E, D, C, B, A and using the fist-matching-rule tie breaker.By introducing resolve filters to break all cyclic requirements imposed by the desired classification policy, we can order any set of rules so that the first-matching-rule tie breaker implements the desired classification policy. Hari et al. [9] "show" that determining the minimum number of resolve filters required to implement a desired classification policy using the first-matching-rule tie breaker is NP-hard. Although this result is correct, the proof provided in [9] is not. In Section 2 we provide a correct proof of this result.

In light of the preceding NP-hard result, Hari et al. [9] propose the addition of a resolve filter for every pair of conflicting filters. When this is done, every packet has a unique most-specific rule that matches it (we assume that the router table is such that every packet is matched by at least one rule in the table) and so any classification policy can be realized. Toward this end of adding a resolve filter for every pair of conflicting filters, Hari et al. [9] propose the use of 2-dimensional 1-bit tries to detect all conflicts when the filters are 2-dimensional prefix filters. Two modes of operation are possible for conflict detection:

1. **Static** $\cdots$ Given a set of n filters, report all pairs of conflicting filters. The static mode is useful "to analyze existing filter databases in firewalls and QoS aware routers to detect conflicts" [9].

2. **Dynamic** $\cdots$ Given a set of n filters and a new filter f, report all conflicts between f and the initial set of n filters. Additionally, add or remove a filter from the given filter set. The dynamic mode is useful "in next generation signaling programs which will carry filters and related packet handling information" [9]. In this application, "as the signalling information propagates through the network, network routers can use the algorithm to report conflicts back to the originators of the signalling requests" [9]. When there are no conflicts, the packet's filters and associated signalling information can be added to the router table.

Using 1-bit 2-dimensional tries augmented with switch pointers, all pairs of conflicting 2-dimensional prefix-filters may be reported in $O(nW + s)$ time, where W is the length of the longest prefix ($W \leq 32$ in IPv4 and $W \leq 128$ in IPv6) and s is the number of pairs of conflicting filters [9]. This time doesn't include the $O(nW^2)$ time need to construct the 2-dimensional trie with switch pointers. For the dynamic version, Hari et al. [9] propose the use of 1-bit 2-dimensional tries without switch pointers. Now, all conflicts between a filter f and a given set of n filters may be reported in $O(W^2 + k)$ time, where k is the number of conflicts between f and the given set of n filters. This time doesn't include the one-time cost of $O(nW)$ to create the 2-dimensional trie structure for the n filters. Also, a filter may be added or removed from the filter set in $O(W)$ time.

Baboescu and Varghese [1, 2] propose a number of bit-vector schemes for conflict detection among prefix filters. However, they use a different notion of conflict than used in [9] and in this paper. They include the cases $f \subset g$ and $g \subset f$ as a conflict between f and g. We refer to this type of conflict as *containment conflict*. The algorithms of Baboescu and Varghese [1, 2] are adapated easily to the conflict model used in this paper. The best of the Baboescu and Varghese algorithms for static conflict detection [2] runs in $O(n^2)$ time and requires $O(n^2)$ space. This algorithm employs compressed binary tries (one for each dimension of a filter) and n-bit vectors, where n is the number of filters. This algorithm may be used for dynamic conflict detection as well. In this case, it takes $O(nW)$ time to report all conflicts between a new filter and an existing set of n filters. A filter may be deleted in $O(W)$ time. A new filter may be inserted in $O(W)$ time if the bit vectors employed by the scheme of [2] are long enough to accommodate an additional filter; otherwise it would take $O(n^2)$ time to resize all the bit vectors to accommodate the new filter. It is important to note that the bit-vector scheme of [2] works for d-dimensional filters, $d > 1$ (and not just for the case $d = 2$). For general d, the complexities stated in this paragraph need to be multiplied by d.

Eppstein and Muthukrishnan [6] develop an $O(n^{3/2})$ algorithm to determine whether or not a set of n range filters has a conflict. They do not explicitly consider reporting all pairwise conflicts in a given filter database.

After providing, in Section 2, a correct proof for Hari et al.'s [9] assertion that determining the minimum number of resolve filters required to implement a desired classification policy using the first-matching-rule tie breaker is NP-hard, we develop a plane-sweep method for the static version of the conflict reporting problem for 2-dimensional prefix filters (Section 3). Our plane-sweep method reports all pairs of conflicting filters in $O(n \log n + s)$ time and uses $O(n)$ space. In Section 5 we introduce the notion of an essential resolve filter and develop an algorithm to determine the essential resolve filters for a given set of 2-dimensional prefix filters. Experimental results are presented in Section 7.

2 NP-hard Proof

Let F be a filter set and let P be a desired policy. Let $MRF(F, P)$ be a minimum set of resolve filters required to implement P using the first-matching-rule tie breaker and let $|MRF(F, P)|$ be the number of filters in $MRF(F, P)$. Let $MRFP$ be the problem of determining $|MRF(F, P)|$.

Hari et al. [9] show how to construct a directed graph $H(F, P)$ in which each vertex represents a filter of F and each directed edge (f, g) denotes the requirement that filter f precede filter g in the router table. Clearly edge (f, g) need be introduced into $H(F, P)$ only when filters f and g conflict and the policy P requires that filter f be selected over filter g for packets that fall in the overlap region of f and g. We shall refer to $H(F, P)$ as the **conflict resolution graph** for (F, P). Hari et al. [9] correctly observe that $|MRF(F, P)|$ equals the size of the smallest feedback arc set[5]

[5]A subset A of the edges of a directed graph G is a **feedback arc set** of G iff the removal from G of the edges in A leaves behind an acyclic graph. The size of the feedback arc set is the number of edges in the feedback arc

of $H(F, P)$ and that adding resolve filters corresponding[6] to the edges in a minimum feedback arc set of $H(F, P)$ is sufficient to realize P using the first-matching-rule tie breaker. From this observation and the fact that determining the size of a minimum feedback arc set is NP-hard [10], Hari et al. [9] conclude that $MRFP$ is NP-hard. This reasoning is incomplete because the only conclusion one can draw from the observation made in [9] is that $MRFP$ is no harder than $MFBASP$ (the problem of determining the size of the smallest feedback arc set of a directed graph). A correct proof that $MRFP$ is NP-hard must show how to solve, in polynomial time, some known NP-hard problem using a polynomial-time algorithm for $MRFP$; not how to solve $MRFP$ in polynomial time using a polynomial time algorithm for a known NP-hard problem.

Theorem 1 *$MRFP$ is NP-hard.*

Proof We show how a polynomial time algorithm for $MRFP$ enables the solution of $MFBASP$ in polynomial time. From this demonstration and the fact that $MFBASP$ is NP-hard it follows that $MRFP$ is NP-hard. The following proof explicitly considers only 2-dimensional prefix filters and therefore only shows that $MRFP$ is NP-hard for 2-dimensional prefix filters. The NP-hardness of $MRFP$ for 2-dimensional range filters as well as for higher dimensional filters in which at least 2 fields are specified as ranges (or prefixes) is an immediate consequence of NP-hardness for 2-dimensional prefix filters.

Let G be a directed graph that represents an arbitrary instance of $MFBASP$. We show how to construct, in polynomial time, an instance (F, P) of $MRFP$ such that $|MRF(F, P)| = |MFBAS(G)|$ ($MFBAS(G)$ is a minimum feedback arc set of G). Hence using a polynomial-time algorithm for $MRFP$ and our construction, we can determine $|MFBAS(G)|$ in polynomial time. The filter set F is obtained in the following way:

set.

[6]For the edge (f, g) the corresponding resolve filter is the resolve filter for f and g.

1. Let n be the number of vertices in the $MFBASP$ instance G and let e be the number of edges in G. F has the $n+e$ 2-dimensional prefix filters $(i,*)$, $1 \leq i \leq n$ and $(*,q)$, $1 \leq q \leq e$. The filter $(i,*)$ matches all packets whose source address is i; the second field of the filter is the destination address field and has been wildcarded. The filter $(i,*)$ represents vertex i of the graph G and the filter $(*,q)$ represents edge q. Notice that the vertex filters are pairwise disjoint (i.e., have no overlap) and that the edge filters also are pairwise disjoint. The conflicting filter pairs of F are $((i,*),(*,q))$, $1 \leq i \leq n$, $1 \leq q \leq e$.

2. Let edge q be (u_q, v_q) (i.e., edge q starts at vertex u_q and terminates at vertex v_q). For each edge (u_q, v_q), $1 \leq q \leq e$, the policy P requires that in the overlap region between $(*,q)$ and $(u_q,*)$ filter $(u_q,*)$ be selected and that filter $(*,q)$ be selected in the overlap region between $(v_q,*)$ and $(*,q)$. For the remaining overlap regions, the conflict may be resolved by selecting either of the overlapping filters[7].

It is easy to see that the (F,P) constructed above is a valid instance of $MRFP$. For every edge (u_q, v_q) of G, the policy P together with the first-matching-rule tie breaker imply that $((u_q,*),(*,q))$ and $((*,q),(v_q,*))$ are edges in the conflict resolution graph $H(F,P)$ for (F,P). Hence $H(F,P)$ may be obtained from G by the following transformation:

1. Into edge (u_q, v_q) of G introduce the vertex $(*,q)$ (this essentially splits the edge (u_q, v_q) into two edges $(u_q,(*,q))$ and $((*,q),v_q)$), $1 \leq q \leq e$.

2. Replace vertex label i with the label $(i,*)$, $1 \leq i \leq n$.

So $|MFBAS(G)| = |MRF(F,P)|$. ■

[7] In case don't cares are not permitted in the policy P, we can augment the filter list F with resolve filters for the remaining overlap regions.

3 Reporting All Conflicts For 2-D Filters

3.1 Resolve Filters and Conflict-Free Filter Sets

Let p be a packet and let $M(p, F)$ be the subset of filters of F that match p. The filter set F is **conflict free** under the most-specific-matching-rule tie breaker iff for every packet p either $|M(p, F)| = 0$ or $M(p, F)$ contains a filter that is more specific than any other filter in $M(p, F)$.

Since $MFRP$ is NP-hard, Hari et al. [9] propose adding to F a resolve filter for every pair of conflicting filters and then using the most-specific-matching-rule tie breaker to implement the desired policy. For this strategy to work, $F \cup resolve(F)$, where $resolve(F)$ is the set of resolve filters for the conflicting pairs of F, must be conflict free. In Theorem 2 we show that $F \cup resolve(F)$ is conflict free for 2-dimensional prefix filters and that it may not be conflict free for 2-dimensional range filters. Hence the strategy proposed in [9] to make F conflict free works for prefix filters but not for range filters. First we introduce some notation.

Let f be a 2-dimensional filter. $X(f)$ and $Y(f)$ are, respectively, the projections of f onto the x and y axes. For example, when $f = (10*, 1101*)$ then $X(f) = 10*$ and $Y(f) = 1101*$ and when $f = ([3, 7], [1, 9])$ then $X(f) = [3, 7]$ and $Y(f) = [1, 9]$. Since a filter defines a set of matching packets (or packet headers), we may use set notation when dealing with filters and their projections. So, for example, $f \cap g$ denotes the set of packets matched by both f and g; alternatively $f \cap g$ denotes the overlap region of f and g; and filters f and g conflict iff $f \cap g \neq \emptyset$, $f \cap g \neq f$, and $f \cap g \neq g$.

Lemma 1 *For every pair of conflicting 2-dimensional filters f and g, $X(f \cap g) = X(f) \cap X(g)$ and $Y(f \cap g) = Y(f) \cap Y(g)$.*

Proof Straightforward. ∎

From Lemma 1, it follows that the resolve filter for f and g is $(X(f) \cap X(g), Y(f) \cap Y(g))$.

Lemma 2 *[Hari et al. [9]] Two prefix filters f and g conflict iff one of the following is true*

1. $X(f) \subset X(g)$ *and* $Y(f) \supset Y(g)$.

2. $X(f) \supset X(g)$ *and* $Y(f) \subset Y(g)$.

Theorem 2 *Let F be a set of 2-dimensional filters.*

1. *When the filters in F are prefix filters, $F \cup resolve(F)$ is conflict free.*

2. *When the filters in F are range filters, $F \cup resolve(F)$ may not be conflict free.*

Proof First consider the case of prefix filters. We prove that in $F \cup resolve(F)$ there is a resolve filter for every pair of conflicting filters. From this and the observation that a resolve filter is more specific than each of the conflicting filters it is the resolve filter for, it follows that $F \cup resolve(F)$ is conflict free. Let f and g be two filters of $F \cup resolve(F)$ that conflict. There are three cases to consider.

Case 1 $f \in F$ and $g \in F$. By definition, the resolve filter $f \cap g$ is in $F \cup resolve(F)$.

Case 2 (a) $f \in F$ and $g \in resolve(F)$ or (b) $g \in F$ and $f \in resolve(F)$. Since (a) and (b) are symmetric, we explicitly consider (a) only. Let g be the resolve filter for g_1 and g_2, where $g_1 \in F$ and $g_2 \in F$. Since f and g conflict, it follows from Lemma 2 that either $X(f) \subset X(g)$ or $X(g) \subset X(f)$. Suppose the former is the case; the latter possibility is symmetric. Also since g_1 and g_2 conflict, Lemma 2 implies that either $X(g_1) \subset X(g_2)$ or $X(g_2) \subset X(g_1)$. Again we consider the former case; the latter case is symmetric. So $g = (X(g_1), Y(g_2))$. Since $X(g) = X(g_1)$, $X(f) \subset X(g_2)$. Also, $Y(g_2) = Y(g) \subset Y(f)$. From Lemma 2 it follows that f and g_2 are in conflict. So the resolve filter $f \cap g_2$ is in $resolve(F)$. Since $f \cap g_2 = (X(f), Y(g_2)) = (X(f), Y(g))$ $= f \cap g$, $f \cap g_2$ is also a resolve filter for f and g.

Case 3 $f \in resolve(F)$ and $g \in resolve(F)$. From Lemma 2, either $X(g) \subset X(f)$ or $X(f) \subset X(g)$. We consider the former case; the latter is symmetric. Let f be the resolve filter for f_1

and f_2, where $f_1 \in F$ and $f_2 \in F$ and let g be the resolve filter for g_1 and g_2, where $g_1 \in F$ and $g_2 \in F$. From Lemma 2 it follows that either $X(f_1) \subset X(f_2)$ or $X(f_2) \subset X(f_1)$ and either $X(g_1) \subset X(g_2)$ or $X(g_2) \subset X(g_1)$. Suppose that the former of each of these possibilities is the case. The remaining cases are symmetric. So $Y(f_2) = Y(f) \subset Y(g) \subset Y(g_1)$, and $X(f_2) \supset X(f_1)$ $= X(f) \supset X(g) = X(g_1)$. Therefore f_2 and g_1 conflict and the resolve filter $f_2 \cap g_1$ is in $resolve(F)$. Since $f_2 \cap g_1 = (X(g_1), Y(f_2)) = (X(g), Y(f)) = f \cap g$, $f_2 \cap g_1$ is also the resolve filter for f and g.

The preceding three cases establish the theorem for prefix filters. Next consider the case of range filters. Figure 2 shows three pairwise conflicting range filters, f, g, and h. The resolve filters are $f \cap g$, $g \cap h$, and $f \cap h$. As can be seen, the filter set $F \cup resolve(F) = \{f, g, h, f \cap g, g \cap h, f \cap h\}$ isn't conflict free. ■

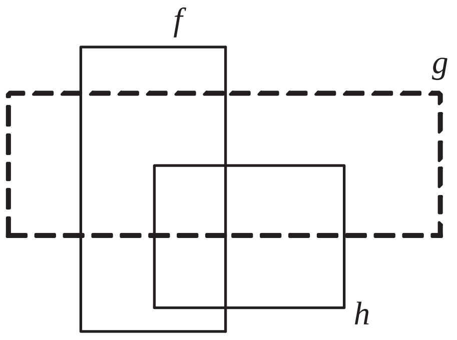


Figure 2: Three pairwise conflicting range filters

3.2 Computing $resolve(F)$

Hari et al. [9] use Lemma 2 to compute $resolve(F)$. Two 2-dimensional 1-bit tries are used. In the first, a 1-bit trie (called the top-level trie) is constructed using the first field $X()$ (say, source field) of the filters of F. Each node x of this 1-bit trie contains a (possibly empty) 1-bit trie constructed from the second field $Y()$ (say, destination field) of those filters of F whose first field corresponds to node x. The tries within each node of the top-level trie are called **bottom-level tries**. In the second 2-dimensional 1-bit trie, the top-level trie is constructed using the destination field of the filters and the bottom-level tries employ the source field. For any given filter f, we can search the first 2-dimensional trie and report all filters $g \in F$ that satisfy condition 1 of Lemma 2. When the 2-dimensional trie is augmented with switch pointers, the time required for this search is $O(W + k1)$, where $k1$ is the number of filters that satisfy condition 1 of Lemma 2. All filters $g \in F$ that satisfy condition 2 of Lemma 2 can be found in $O(W + k2)$ time, where $k2$ is the number of filters that satisfy condition 2 of Lemma 2, by searching the second trie augmented with switch pointers. We may compute $resolve(F)$ by repeating the just described searches for every $f \in F$. The time required, exclusive of the time required to construct the two 2-dimensional tries with switch pointers is $O(nW + s)$, where n is the number of filters in F and $s = |resolve(F)|$. The switch pointers needed for conflict detection may be constructed in $O(nW^2)$ time.

In this section, we develop a plane-sweep algorithm to compute $resolve(F)$ in $O(n \log n + s)$ time, where $s = |resolve(F)|$. Our plane-sweep algorithm doesn't employ Lemma 2. Rather, it works by detecting orthogonal line crossings.

Two line segments **cross** iff they are orthogonal and share a common point. Two line segments **perfectly cross** iff they cross and the crossing point is not an endpoint of either line segment. If two line segments cross but do not cross perfectly, the crossing is called an **imperfect crossing**. Figure 3 shows all the cases for an imperfect crossing.

The 2-dimensional filter f is a *nontrivial filter* iff it is not a rectilinear line segment (a 2-dimensional point is a special case of a line segment). Otherwise, f is a *trivial filter*. For example, $(11*, 1001)$ is a trivial filter, but $(11*, 100*)$ is a nontrivial filter.

Figure 4 shows two examples in which two nontrivial prefix filters conflict.

Theorem 3 *Two nontrivial prefix filters f and g conflict iff an edge of f perfectly crosses an*

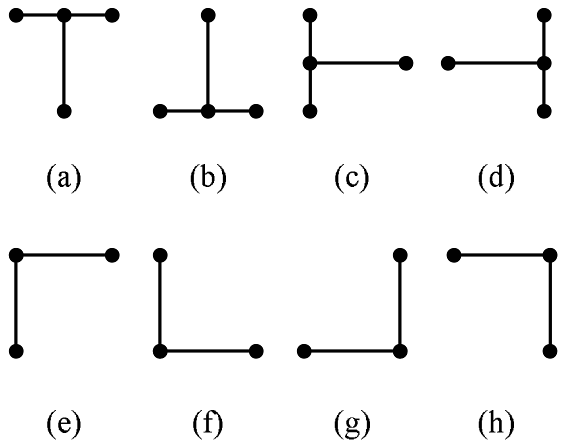


Figure 3: Imperfect crossings

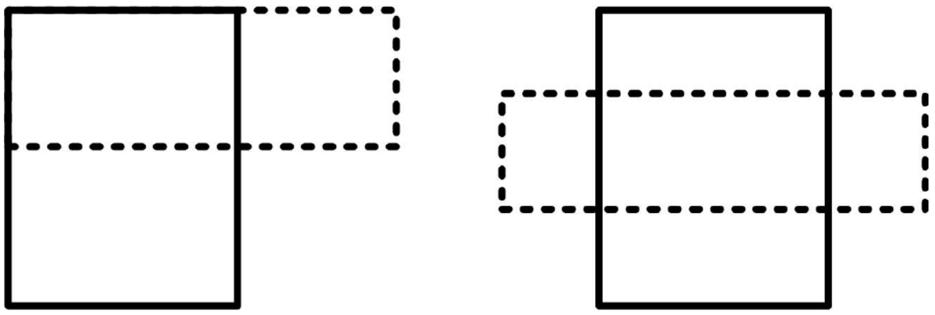


Figure 4: Examples of conflicting nontrivial prefix filters

edge of g.

Proof It is easy to see that $f \cap g \neq \emptyset$ and neither f nor g is more specific than the other if an edge of f perfectly crosses an edge of g.

Now we show that an edge of f must perfectly cross some edge of g whenever f and g conflict. Since f and g conflict, either condition 1 or condition 2 of Lemma 2 is true. Assume condition 1 is true (the case when condition 2 is true is symmetric). Since $Y(f) \supset Y(g)$, the bottom edge of f is either below the bottom edge of g or has the same y value as does the bottom edge of g. In either case, there must be a perfect crossing since $X(f) \subset X(g)$. ■

Since F may contain trivial filters, we define an operation, mag (magnify) that converts all filters (trivial and nontrivial) into nontrivial filters. This operation preserves filter conflicts. Let filter $f = ([x_1, x_2], [y_1, y_2])$, then $mag(f) = ([x_1 0, x_2 1], [y_1 0, y_2 1])$, where $x_1 0$ is x_1 concatenated with bit 0.

Notice that when $x_1 = x_2$, f is a vertical line segment. $mag(f)$ transforms a line segment into a nontrivial rectangle. For example, if $f = ([8, 8], [8, 11])$, then $mag(f) = ([16, 17], [16, 23])$ and if $g = ([8, 11], [8, 9])$, then $mag(g) = ([16, 23], [16, 19])$. Figure 5 shows these two filters and the corresponding filters after applying the mag operator. We not only convert a trivial filter to a nontrivial filter by applying the mag operator, but also get a perfect crossing, which can be used to detect the conflict between these two filters.

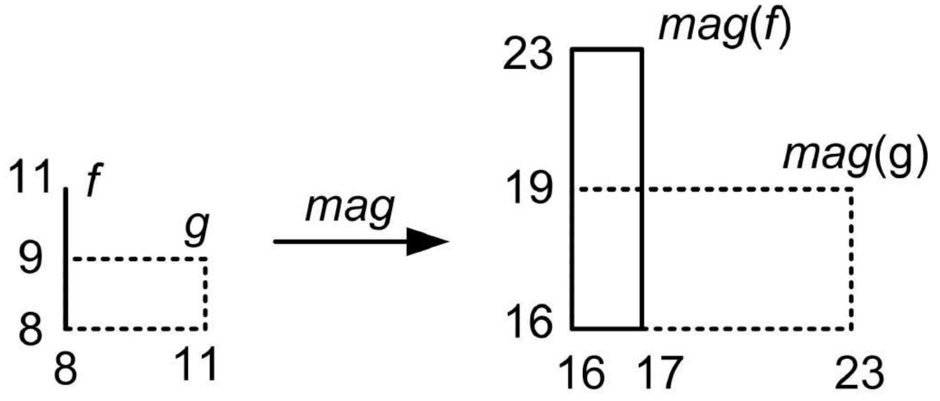


Figure 5: Filters $([8, 8], [8, 11])$ and $([8, 11], [8, 9])$ and the corresponding filters after applying mag.

Lemma 3 *The following are true.*

1. $[a, b] \cap [c, d] \neq \emptyset$ *iff* $[a0, b1] \cap [c0, d1] \neq \emptyset$.

2. $[a, b] \subset [c, d]$ *iff* $[a0, b1] \subset [c0, d1]$.

Proof It is easy to see that $a0 < c0$ iff $a < c$, and $a1 < c0$ iff $a < c$. For the first claim of the lemma, suppose that $[a, b] \cap [c, d] \neq \emptyset$. If $[a0, b1] \cap [c0, d1] = \emptyset$, then either $b1 < c0$ or $d1 < a0$.

So either $b < c$ or $d < a$. In either case, we have $[a, b] \cap [c, d] = \emptyset$, a contradiction. Next suppose that $[a0, b1] \cap [c0, d1] \neq \emptyset$. If $[a, b] \cap [c, d] = \emptyset$, then either $b < c$ or $d < a$. So either $b1 < c0$ or $d1 < a0$. In either case, we have $[a0, b1] \cap [c0, d1] = \emptyset$, a contradiction.

It is easy to see that the second claim of the lemma is true because $c \leq a \leq b \leq d$ iff $c0 \leq a0 < b1 \leq d1$. ■

Theorem 4 *Prefix filters f and g conflict iff $mag(f)$ and $mag(g)$ conflict.*

Proof Follows from Lemmas 2 and 3. ■

From Theorems 3 and 4, it follows that we can identify all pairs of filters in F that are in conflict by first applying the *mag* operator on each filter of F to obtain $mag(F)$ and then determining all perfect crossings of the rectilnear line segments of $mag(F)$. The latter can be done in $O(n \log n + s)$, where s is the number of conflicts or the number of perfect crossings, time and $O(n)$ space using Bentley and Ottmann's [4] plane-sweep method.

4 Online Conflict Detection and Reporting

From Theorems 3 and 4, it follows that we can identify all filters in F that are in conflict with a new filter f by first applying the *mag* operator on f and each filter of F to obtain $mag(f)$ and $mag(F)$ and then determining all perfect crossings between vertical (horizontal) line segments of $mag(f)$ and horizontal (vertical) line segments of $mag(F)$. The latter problem can be solved using orthogonal line segment intersection reporting algorithms, i.e., to report all the segments in a finite set of horizontal line segments that intersect any vertical line segment. Both static and dynamic versions of orthogonal line segment intersection reporting have been extensively studied. Mortensen [17] gives a fully-dynamic solution that supports update in $O(\log n)$ time and intersection reporting in $O(\log n + s)$, where s is the number of intersections. The solution of Mortenson [17] requires $O(n \log n/ \log\log n)$ space. It immediately follows that all conflicts between f and F can be reported in $O(\log n + s)$ time, where s is the number of conflicts, using $O(n \log n/ \log\log n)$ space and that we can update the data structure (i.e., add a new filter or remove an old filter) in $O(\log n)$ time. By comparison, the 1-bit 2-dimensional tries scheme of [9] reports all conflicts between f and F in $O(W^2 + s)$ time and takes $O(W)$ time to do an update. The space required by the scheme of [9] is $O(nW)$. The bit-vector scheme of Baboescu and Varghese [2], takes $O(nW)$ time to report all conflicts between a new filter and an existing set of n filters. A filter may be deleted in $O(W)$ time. A new filter may be inserted in $O(W)$ time if the bit vectors employed by the scheme of [2] are long enough to accommodate an additional filter; otherwise it would take $O(n^2)$ time to resize all the bit vectors to accommodate the new filter.

5 Essential Resolve Filters

The rationale for adding the filters in $resolve(F)$ to F is to arrive at a set of filters that is free of conflict (i.e., for every packet being classified, there is either no matching filter or there is a unique most-specific matching-filter). This objective, however, can often be met by adding to F only a subset of the filters in $resolve(F)$. For example, suppose that f and g are conflicting filters of F. If the resolve filter $f \cap g$ already is in F or if F contains a set of filters whose union equals $f \cap g$, then we can avoid adding $f \cap g$ to F.

A filter $f \in resolve(F)$ is an **essential resolve filter** iff $F \cup resolve(F) - \{f\}$ has no subset whose union equals f. Let $essential(F)$ be the set of essential resolve filters of F.

Theorem 5 *For every set F of prefix filters, $F \cup essential(F)$ is conflict free.*

Proof Follows from Theorem 2 and the observation that the removal of non-essential filters of $resolve(F)$ from $F \cup resolve(F)$ doesn't affect the conflict-free property. ■

Our algorithm to determine whether or not a resolve filter is essential employs the following lemma.

Lemma 4 *Let a, b and c be prefix filters and let $c = [s_1s_2...s_i*, d_1d_2...d_j*]$. $a \cup b = c$ iff one of the following is true.*

1. $a = c \wedge b \subseteq c$.

2. $a \subseteq c \wedge b = c$.

3. $\{a, b\} = \{[s_1s_2...s_i0*, d_1d_2...d_j*], [s_1s_2...s_i1*, d_1d_2...d_j*]\}$.

4. $\{a, b\} = \{[s_1s_2...s_i*, d_1d_2...d_j0*], [s_1s_2...s_i*, d_1d_2...d_j1*]\}$.

Proof Straightforward. ■

We first construct a 2-dimensional trie T for $F \cup resolve(F)$. The top-level trie of T is based on $X(f)$ and the bottom-level trie is based on $Y(f)$. Then, we invoke $TopTrieTraversal(root(T))$ (Figure 6) to transform T into a 2-dimensional trie for the filter set $C = combine(F \cup resolve(F))$, which comprises all 2-dimensional prefix filters that are the union of some subset of the filters of $F \cup resolve(F)$. Function $TopTrieTraversal$ is a recursive function that first does this transformation recursively on the left and right top-level subtries of $root(T)$ and then applies Lemma 5 to complete the transformation at $root(T)$. Function $trie(z)$ returns the bottom-level trie that is in node z of the top-level trie if z is not *null*, and returns *null* otherwise. Function $hasFilter(x)$ returns *true* if there is a filter (either an original filter of T or one added during the transformation into the trie for C) associated with node x of the bottom-level trie, and returns $false$ otherwise. The function $setHasFilter(x,\ truthValue)$, sets the $hasFilter$ value of node x to $truthValue$. So, the function $hasFilter(x)$ returns *true* when executed following an invocation of $setHasFilter(x,\ true)$. Function $TopTrieTraversal$ performs a postorder traversal on the top-level trie. Function $BottomTrieTraveral$ performs a postorder traversal on a bottom-level trie. Function $BottomTriesTraveral(xNode, yNode, zTrie)$ performs a synchronized postorder traversal on two bottom-level tries that are in the two children of the top-level trie node z, and $zTrie = trie(z)$ is the bottom-level trie pointed at by node z. We use the abbreviations TTT, BTT, and $BTST$ for the three functions of Figure 6.

Table 1 gives a set of six prefix filters. Figure 7(a) shows the 2-dimensional trie for $F \cup resolve(F)$. After invoking $TopTrieTraversal(root(topLevelTrie))$, the 2-dimensional trie of Figure 7(a) is transformed into the trie of Figure 7(b). The dark shaded nodes represent nodes into which a new filter was added during the transformation process. The numbers next to these dark shaded nodes give the order by which these node are generated by the transformation traversal.

Filter	Source Prefix	Destination Prefix
a	000*	00*
b	001*	00*
c	00*	01*
d	01*	0*
e	0*	1*
f	1*	*

Table 1: A set of prefix filters

Lemma 5 *Let T be the 2-dimensional trie for the prefix filter set G. The invocation $TTT(root(T))$ transforms T into the 2-dimensional trie for $combine(G)$.*

Proof We prove this by induction on the number of levels in T. When T has 0 levels (i.e., T is empty), $G = combine(G) = \emptyset$ and $TTT(null)$ leaves T as an empty trie, which is the trie for $combine(G)$. Assume that the lemma is true for all T that have up to l levels. For the induction step consider a T with $l + 1$ levels. Let G be the filters in T, G_L the filters in the left subtrie

```
**Algorithm** $TopTrieTraversal(node)\{$
    **if** $(node)\{$
        $TopTrieTraversal(node.left);$
        $TopTrieTraversal(node.right);$
        $BottomTriesTraveral(root(trie(node.left)), root(trie(node.right)), trie(node));$
                // Case 3 of Lemma 4
        $BottomTrieTraveral(root(trie(node)));$
                // Case 4 of Lemma 4
    }
}

**Algorithm** $BottomTrieTraveral(node)\{$
    **if** $(node)\{$
        $BottomTrieTraveral(node.left);$
        $BottomTrieTraveral(node.right);$
        **if** $(hasFilter(node.left) \wedge hasFilter(node.right))$
            $setHasFilter(node, true);$
    }
}

**Algorithm** $BottomTriesTraveral(xNode, yNode, zTrie)\{$
    **if**$(xNode \wedge yNode)\{$
        $BottomTriesTraveral(xNode.left, yNode.left, zTrie);$
        $BottomTriesTraveral(xNode.right, yNode.right, zTrie);$
        **if**$(hasFilter(xNode) \wedge hasFilter(yNode))$
            Insert filter $filter(xNode) \cup filter(yNode)$ into $zTrie$;
    }
}
```

Figure 6: Combine Filters

of $root(T)$ and G_R the filters in the right subtrie of $root(T)$. From the induction hypothesis it follows that the invocations $TTT(root(T).left)$ and $TTT(root(T).right)$, transform the left and right subtries of $root(T)$ into the tries for $combine(G_L)$ and $combine(G_R)$. We observe that the

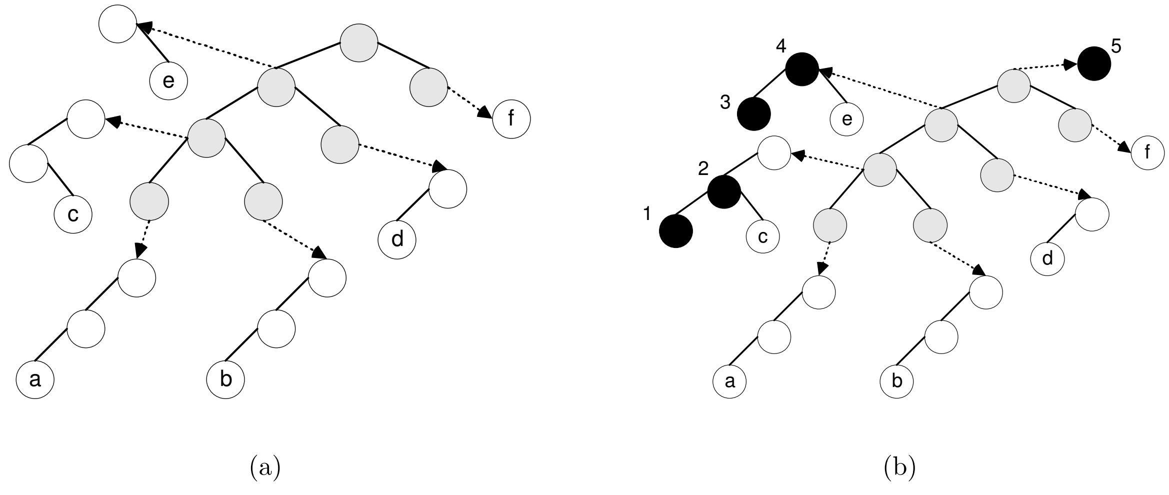


Figure 7: The 2D trie for $F \cup resolve(F)$ and $combine(F \cup resolve(F)$

only additional filters in *combine*(G) are those that belong in *trie*$(root(T))$. From Lemma 4 these additional filters are either already in *root*$(trie(T))$ (original filters of G that are in *trie*$(root(T))$ as well as cases 1 and 2 of the lemma) or are obtainable by combining two filters as in cases 3 and 4 of the lemma. The invocation of $BTST$ adds to *trie*$(root(T))$ all filters constructable as in case 3 of the lemma; the ensuing invocation of $BTT(trie(root(T))$ handles case 4 of the lemma. ■

Lemma 6 *Let G be a set of 2-dimensional prefix filters and let $m = |G|$. $|combine(G)| = O(mW)$, where W is the length of the longest prefix in any field of the filters of G.*

Proof Let T be the 2-dimensional trie for G and let T' be that for *combine*(G). T' is the 2-dimensional trie computed by the invocation $TTT(root(T))$. The top-level trie of T has at most $W + 1$ levels. Let m_i be the total number of filters of G that are stored in the level-i bottom-level tries (i.e., the bottom-level tries that are in nodes at level i of the top-level trie). Note that $\sum_{i=0}^{W} m_i = m$. Let m'_i be the total number of filters of *combine*(G) that are stored in the level-i bottom-level tries of T'. Note that $\sum_{i=0}^{W} m'_i = |combine(G)|$.

Suppose that *node* is a leaf of the top-level trie T and that the number of filters of G that are stored in the bottom-level trie $\mathit{trie}(\mathit{node})$ is r. The invocation $TTT(\mathit{node})$ makes the invocations $TTT(\mathit{null})$, $TTT(\mathit{null})$, $BTST(\mathit{null}, \mathit{null}, \mathit{trie}(\mathit{node}))$ and $BTT(\mathit{root}(\mathit{trie}(\mathit{node})))$. The first three of these invocations make no change in the number of filters stored in any node of T. The fourth invocation may combine the filters stored in the children of a degree 2 node x of $\mathit{trie}(\mathit{node})$ and store the combined filter into node x. This together with the observation that the number of degree 2 nodes in $\mathit{trie}(\mathit{node})$ is less than r implies that the number of new filters in $\mathit{trie}(\mathit{node})$ following the execution $BTT(\mathit{root}(\mathit{trie}(\mathit{node})))$ is less than r. Hence the total number of filters in $\mathit{trie}(\mathit{node})$ of T' is less than $2r$ (new and original). Since all nodes at level W of the top level trie are leaves (we assume without loss of generality that there is at least one node at level W), $m'_W < 2m_W$.

Next consider the level i, $i < W$, nodes of the top-level trie. These nodes together have m_i filters to begin with. The invocations of $BTST$ made from these level i nodes may add at most $m'_{i+1}/2$ filters to the level i bottom-level tries (each filter added to $\mathit{trie}(\mathit{node})$ by $BTST$ is the combination of 2 filters that are in level $i+1$ bottom-level subtries; no filter in a level $i+1$ bottom-level subtrie may contribute to more than 1 filter added to $\mathit{trie}(\mathit{node})$). So, excluding the filters added by the BTT invocations made at level i, the number of filters in the level i bottom-level tries of T' is at most $m_i + m'_{i+1}/2$. As was the case for leaf nodes, the number of filters added by the BTT invocations is less than the number of filters in the level i bottom-level tries prior to the BTT invocations. Hence,

$$\begin{aligned} m'_i &< 2(m_i + m'_{i+1}/2) \\ &= 2m_i + m'_{i+1} \\ &< 2m_i + 2m_{i+1} + m'_{i+2} \\ &< 2\sum_{j=i}^{W} m_j \\ &\leq 2m \end{aligned}$$

Therefore, $|combine(G)| = \sum_{i=0}^{W} m'_i < \sum_{i=0}^{W} 2m = 2m(W+1) = O(mW)$. ■

The complexity of $TopTrieTraversal$ is $O(mW)$, where m is the number of filters in the initial filter set G. This follows from the observation that the total number of nodes in the top- and bottom-level tries is $O(mW)$ and the traversals take time linear in the number of nodes in the tries being traversed. Note that no bottom-level trie is traversed more than twice (once during the execution of BTT and once for $BTST$).

To determine $essential(F)$, we first compute $resolve(F)$ using the plane-sweep method as in Section 3. Then we construct the 2-dimensional trie for $G = F \cup resolve(F)$ and run $TopTrieTraversal$. While $TopTrieTraversal$ is computing $combine(G)$ we can detect the non-essential filters of $resolve(F)$ (when a filter g is added to $trie(node)$ either by BTT or $BTST$, we check whether g is already in $trie(node)$; if g is already present and is a resolve filter, g is non-essential). It takes $(n \log n + s)$ time to determine $resolve(F)$ from F ($n = |F|$ and $s = |resolve(F)|$) and an additional $O((n+s)W)$ time to construct the 2-dimensional trie for $F \cup resolve(F)$ and execute $TopTrieTraversal$ to identify $essential(F)$.

6 Adaptation of Conflict-Detection Algorithm of [2]

The bit-vector scheme of Baboescu and Varghese [2] employs a slightly different definition of filter conflict than that used by Hari et al. [9] and us. Baboescu and Varghese report filter containment as a conflict whereas Hari et al. do not. Hari et al. [9] state that containment conflicts may be handled by rule ordering or by proper priority assignment. Resolve filters are needed only for conflicts other than containment conflicts. So, Hari et al. [9] report only non-containment conflicts.

To compare the bit-vector scheme of Baboescu and Varghese [2] with the scheme proposed in this paper, we need to modify the bit-vector scheme slightly so as to conform to the conflict model used in this paper and in [9]. For this adaptation, we also optimize the scheme of Baboescu and Varghese [2] so that the number of bit-vector unions done when detecting all conflicts between a filter g and a set of n filters is $O(1)$ rather than $O(W)$ as in the algorithm proposed in [2].

Let $trie_x$ and $trie_y$, respectively, be the (uncompressed) extended[8] one-bit tries for the x and y fields of the filters in F. Let $prefix(N)$ be the prefix represented by node N of an extended one-bit trie. For example, $prefix(root) = *$, $prefix(root.left) = 0*$, $prefix(root.right) = 1*$ and $prefix(root.left.right) = 01*$. Each node, N, in $trie_x$ ($trie_y$) has two bit vectors, $bv1$ and $bv2$. Each bit vector has n bits, where n is the number of filters in F. Let f_i be the ith filter of F. When $N \in trie_x$ ($N \in trie_y$), the i-th bit of $N.bv1$ is one iff $X(f_i)$ ($Y(f_i)$) is a proper prefix of $prefix(N)$; the i-th bit of $N.bv2$ is one iff $prefix(N)$ is a proper prefix of $X(f_i)$ ($Y(f_i)$). Note that our definitions for $bv1$ and $bv2$ are slightly different from those used by Bobescu and Varghese [2] for the bit-vectors they store in each node.

With the preceding definition of $bv1$ and $bv2$, it is easy to implement Lemma 2 and find all conflicts between a new filter g and a given filter set F. We search $trie_x$ ($trie_y$) for the node N_x (N_y) such that $prefix(N_x) = X(g)$ ($prefix(N_y) = Y(g)$). From the definitions of $bv1$ and $bv2$, it follows that $N_x.bv1 \cap N_y.bv2$ gives all fs in F such that $X(g) \subset X(f)$ and $Y(g) \supset Y(f)$ and $N_x.bv2 \cap N_y.bv1$ gives all fs in F such that $X(g) \supset X(f)$ and $Y(g) \subset Y(f)$. By Lemma 2, the algorithm finds all conflicts between g and F.

Figure 8 gives the corresponding algorithm.

Algorithm $BitVector$ (Figure 8) may be optimized as in [2] by applying path compression to remove single-branch trie nodes N for which $prefix(N) \neq X(f)$ (in case of $trie_x$ and $\neq Y(f)$ in case of $trie_y$) for every $f \in F$ and adding an aggregation bit-vector for each $bv1$ and $bv2$ to avoid reading portions of $bv1$ or $bv2$ that contain only zeroes. When these optimizations are

[8]An extended one-bit trie is obtained from an ordinary one-bit trie by adding an external node wherever the ordinary one-bit trie has an empty subtree.

```
**Algorithm** $BitVector(g, trie_x, trie_y)\{$
    $//trie_x$ and $trie_y$ are extended one-bit tries.
    Let $N_x \in trie_x$ be the last node encountered in a search for $X(g)$.
    Let $N_y \in trie_y$ be the last node encountered in a search for $Y(g)$.
    **return** $(N_x.bv1 \cap N_y.bv2) \cup (N_x.bv2 \cap N_y.bv1)$;
$\}$
```

Figure 8: Bit-vector algorithm to detect conflicts between filter g and a set of filters.

performed, the total number of nodes in $trie_x$ and $trie_y$ are $O(n)$; the time complexity of the optimized version of algorithm $BitVector$ is $O(n)$; and the space complexity is $O(n^2)$. Another optimization, which reduces the space complexity by a constant factor is to eliminate the external nodes. This optimization changes the definition of $bv1$ so that when $N \in trie_x$ ($N \in trie_y$), the i-th bit of $N.bv1$ is one iff $X(f_i)$ ($Y(f_i)$) is a prefix (including equal to) of $prefix(N)$.

To report conflicts between all pairs of filters in a filter set F, we first construct the tries (along with the $bv1$ and $bv2$ bit vectors) for the x and y components of the filters of F and then run algorithm $BitVector$ n times, with g being a different filter of F on each of the n runs. The total time to construct the tries and report the conflicts is $O(n^2)$ and the space required also is $O(n^2)$.

7 Experimental Results

We implemented our plane-sweep conflict detection algorithm of Section 3, the conflict detection algorithm $FastDetect$ of Hari et al. [9] and an optimized version of our customization $BitVector$ of the conflict detection algorithm of Baboescu and Varghese [2] in C++. The optimized version of $BitVector$ incorporated path compression and aggregation as suggested by Baboescu and Varghese [2] and also eliminated external nodes by using the modified definition of $bv1$ given in Section 6. The three algorithms were benchmarked on a 2.4GHz Pentium4 PC that has 1GB of memory. To assess the performance of these three algorithms, we randomly generated 2-dimensional filters using the algorithm of Figure 9[9]. In this algorithm, $max = 2^{32} - 1$ for IPv4 prefixes, the function $random(a, b)$ generates a random integer that is uniformly distributed between a and b, the length of the source address prefix (i.e., first field of the filter) is required to be in the range $[srcLenLow, srcLenHigh]$, and the length of the destination address prefix is required to be in the range $[destLenLow, destLenHigh]$. The expression $src/srcLength$ creates a prefix whose bits are the first $srcLength$ bits of src.

```
src = random(0, max);
srcLength = random(srcLenLow, srcLenHigh);
dst = random(0, max);
destLength = random(destLenLow, destLenHigh);
prefixFilter = (src/srcLength, dst/dstLength)
```

Figure 9: Method to generate a random filter

By repeatedly applying the method of Figure 9 and discarding duplicate filters, we can generate random filter sets F of any desired size (of course, the size is limited by the number of possible distinct filters). When we permit the source and destination prefix lengths to be random numbers in the permissible IPv4 range [0,32] (i.e., `srcLenLow` = `destLenLow` = 0 and `srcLenHigh` = `destLenHigh` = 32), the random filter sets F that are generated have many pairs of conflicting filters. For example, when $|F|$ = 1,000 the number of conflicts is more than 10,000. When the length of the source and destination prefixes is constrained to the range [16, 32] the generated filter sets have no conflicts; and when the range for source prefix length is [0, 32] and that for destination prefix length is [10, 32], the number of conflicts is non-zero and much smaller than $|F|$. We believe that this last choice of length ranges most closely reflects the number of conflicts to be found in real filter databases.

[9]We resorted to generation of random filters because despite significant effort we were unable to get either the large real or synthetic data sets used by others in their research.

For our experiments, we used the filter set sizes 1000, 5000, 10000, 20000 and 30000. For each filter set size and prefix length constraint, ten random filter sets were generated. The memory and time required to compute $resolve(F)$ for each of these 10 filter sets was determined. The mean memory and time requirements are reported in Tables 2 through 4 and in Figures 10 and 11. The reported memory requirement does not include the memory required to store the original filter set or that required to save and return the pairs of conflicting filters. However, the reported times include the time required to dynamically allocate the memory needed to save and return the pairs of conflicting filters.

$\|F\|$		Detection Time (ms)			Memory Requirement(kB)			$\|resolve(F)\|$
		PlaneSweep	FastDetect	BitVector	PlaneSweep	FastDetect	BitVector	
1000	mean	3	53	41	117	926	992	0
	std	7	8	8	0	1	0	0
5000	mean	29	242	255	586	4,354	21,191	0
	std	7	8	7	0	10	19	0
10000	mean	52	495	729	1,172	8,476	82,529	0
	std	8	8	11	0	11	56	0
20000	mean	109	988	2,325	2,344	16,491	323,201	0
	std	0	6	36	0	22	166	0
30000	mean	175	1,477	4,802	3,515	24,338	718,910	0
	std	7	8	67	0	15	366	0

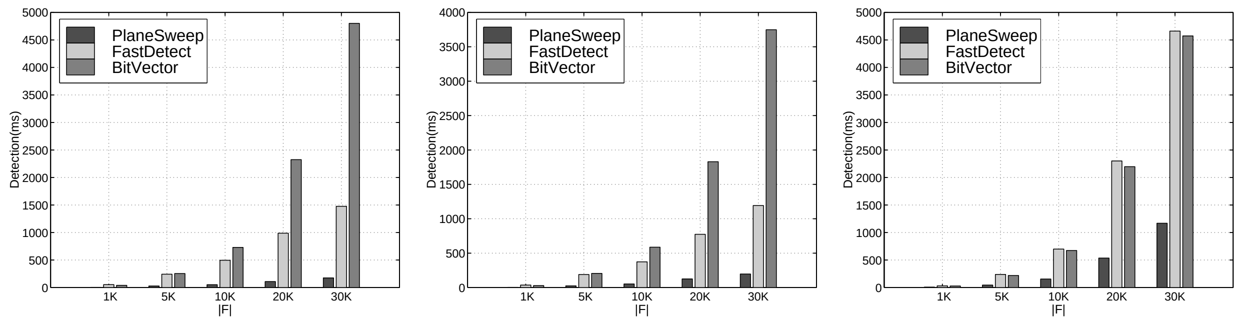
Table 2: Source prefix length range [16, 32], destination prefix length range [16, 32]

$\|F\|$		Detection Time (ms)			Memory Requirement(kB)			$\|resolve(F)\|$
		PlaneSweep	FastDetect	BitVector	PlaneSweep	FastDetect	BitVector	
1000	mean	3	36	29	121	688	834	6
	std	7	8	7	0	13	4	3
5000	mean	25	189	205	602	3,174	16,561	116
	std	8	5	9	1	20	84	11
10000	mean	52	373	587	1,203	6,149	62,401	467
	std	8	5	15	1	30	249	29
20000	mean	125	774	1,829	2,403	11,900	236,319	1,858
	std	0	8	21	2	28	420	78
30000	mean	197	1,193	3,749	3,605	17,497	516,050	4,280
	std	8	7	19	3	73	1,872	165

Table 3: Source prefix length range [0, 32], destination prefix length range [10, 32]

$\lvert F \rvert$		Detection Time (ms)			Memory Requirement(kB)			$\lvert resolve(F) \rvert$
		PlaneSweep	FastDetect	BitVector	PlaneSweep	FastDetect	BitVector	
1000	mean	8	31	28	116	597	737	2,704
	std	8	0	7	0	5	6	221
5000	mean	45	237	219	579	2,755	14,429	67,850
	std	5	7	7	2	11	38	3,051
10000	mean	154	699	673	1,157	5,347	54,200	255,150
	std	5	15	5	2	17	184	9,144
20000	mean	536	2,300	2,197	2,316	10,377	206,115	958,701
	std	7	35	20	2	22	394	24,225
30000	mean	1,169	4,661	4,573	3,474	15,304	452,216	2,052,087
	std	30	87	18	3	55	1,646	56,502

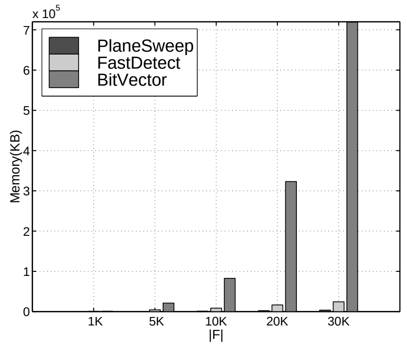
Table 4: Source prefix length range [0, 32], destination prefix length range [0, 32]



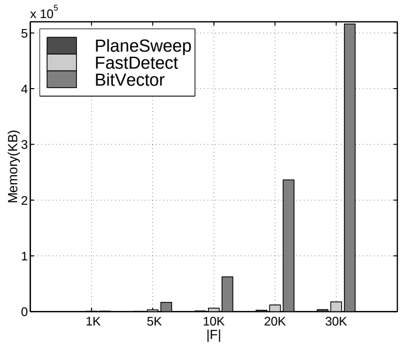
(a) (b) (c)

Figure 10: Time to detect all conflicts. (a) [16, 32] $\times$ [16, 32], (b) [0, 32] $\times$ [10, 32], (c) [0, 32] $\times$ [0, 32]

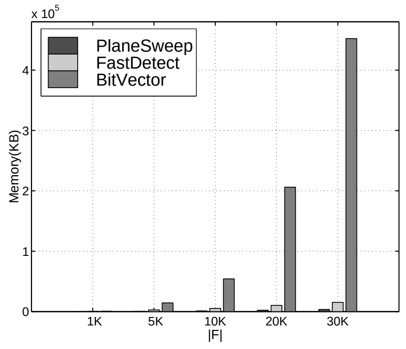
On all of our 15 test sets, our proposed plane-sweep conflict-detection algorithm was noticibly faster than both $FastDetect$ and $BitVector$. Additionally, our algorithm took much less memory. $FastDetect$ takes between 4 and 17 times the time taken by the plane sweep method. It takes also 4 to 8 times the memory required by the plane sweep method. $BitVector$ takes between 4 and 27 times the time taken by the plane sweep method and between 6 and 205 times the memory. Notice that on our 30,000 filter test sets, $BitVector$ requires between 452MB and 719MB of memory, whereas our proposed plane sweep method requires only 3.5MB. On these same tests



(a)



(b)



(c)

Figure 11: Memory requirement. (a) [16, 32] $\times$ [16, 32], (b) [0, 32] $\times$ [10, 32], (c) [0, 32] $\times$ [0, 32]

sets, $FastDetect$ requires between 15MB and 24MB.

Table 5 gives the time and memory required to determine $essential(F)$ for the case in which the source prefix length range is [0, 32] and the destination prefix length range is [10,32]. The reported times include the time needed to construct the 2d trie for $F \cup resolve(F)$. Although it takes 3 to 6 times as much time and about 3 times as much memory to determine $essential(F)$ as taken by the plane sweep method to determine $resolve(F)$, the time and memory needed to determine $essential(F)$ is about one-half that needed by $FastDetect$ to compute $resolve(F)$. For our test data, almost all the conflict pairs in $resolve(F)$ are essential.

8 Conclusion

We have provided a correct proof that $MRFP$ is NP-hard. Additionally, we have developed a fast plane-sweep algorithm to report all filter conflicts in a set of 2-dimensional prefix filters. Our plane-sweep algorithm runs in $O(n \log n + s)$ time where n is the number of filters and s is the number of conflicts. This represents an asymptotic improvement over previously proposed conflict reporting algorithms. Experiments conducted by us reveal that our algorithm is considerably

$\lvert F \rvert$		Run Time (ms)	Memory(kB)	$\lvert resolve(F) \rvert$	$\lvert essential(F) \rvert$
1000	mean	17.1	348.4	4.6	4.6
	std	4.9	2.1	0.5	0.5
5000	mean	97.2	1646.9	110.2	109.8
	std	6.7	5.4	15.8	15.5
10000	mean	175.0	3242.0	471.4	470.5
	std	6.3	13.7	14.8	15.0
20000	mean	373.6	6507.5	1854.4	1849.9
	std	8.7	11.7	50.9	51.6
30000	mean	592.4	9897.3	4178.6	4169.7
	std	5.1	24.1	126.1	125.6

Table 5: Time and memory to determine $essential(F)$; [0, 32] $\times$ [10, 32]

faster, even on practical sized databases, than the conflict reporting algorithms of Hari et al. [9] and Baboescu and Varghese [2]. Additionally, our algorithm requires much less space. Finally, we have introduced the notion of an essential resolve filter and developed an efficient algorithm to compute $essential(F)$, the set of essential resolve filters for the filter set F.

References

[1] F.Baboescu and G.Varghese, Scalable packet classification, *ACM SIGCOMM*, 2001.

[2] F.Baboescu and G.Varghese, Fast and Scalable Conflict Detection for Packet Classifiers, *10th IEEE International Conference on Network Protocols (ICNP'02)*,2002.

[3] F.Baboescu, S.Singh and G.Varghese, Packet Classification for Core Routers: Is there an alternative to CAMs? *INFOCOM*, 2003.

[4] J.L.Bentley and T.A.Ottmann, Algorithms for Reporting and Counting Geometric Intersections, *IEEE Transactions on Computers*, C-28, 9, 1979, 643-647.

[5] M. Buddhikot, S. Suri and M. Waldvogel, Space decomposition techniques for fast layer-4 switching, *Conference on High Speed Networks*, 1998.

[6] D. Eppstein and S. Muthukrishnan, Internet packet filter management and rectangle geometry, *12th ACM-SIAM Symp. on Discrete Algorithms*, 2001, 827-835.

[7] A. Feldman and S. Muthukrishnan, Tradeoffs for packet classification, *INFOCOM*, 2000.

[8] P. Gupta and N. McKeown, Packet classification using hierarchical intelligent cuts, *ACM SIGCOMM*, 1999.

[9] A.Hari, S.Suri, G.Parulkar, Detecting and resolving packet filter conflicts, *INFOCOM*, 2000.

[10] E.Horowitz, S.Sahni, and S.Rajasekeran, Computer Algorithms/C++, W. H. Freeman, NY, 1997.

[11] T. Lakshman and D. Stidialis, High speed policy-based packet forwarding using efficient multi-dimensional range matching, *ACM SIGCOMM*, 1998.

[12] L. Qiu, G. Varghese and S. Suri, Fast firewall implementation for software and hardware based routers. *9th International Conference on Network Protocolos ICNP*, 2001.

[13] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, Survey and taxonomy of IP address lookup algorithms, *IEEE Network*, 2001, 8-23.

[14] S. Sahni, K. Kim, and H. Lu, Data structures for one-dimensional packet classification using most-specific-rule matching, *International Journal on Foundations of Computer Science*, 14, 3, 2003, 337-358.

[15] V. Srinivasan, S. Suri, and G. Varghese, Packet classification using tuple space search, *ACM SIGCOMM*, 1999.

[16] V. Srinivasan, A packet classification and filter management system, *INFOCOM*, 2001.

[17] C. W. Mortensen, Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time, *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, 2003.

[18] C. Kaufman, R. Perlman and M. Speciner, *Network Security: Private communication in a public world*, Second Edition, Chapter 17, Prentice Hall, NJ, 2002.