Fast Packet Classification for Two-Dimensional Conflict-Free Filters

Priyank Warkhede Subhash Suri George Varghese

Abstract—**Routers can use packet classification to support advanced functions such as QoS routing, virtual private networks and access control. Unlike traditional routers, which forward packets based on destination address only, routers with packet classification capability can forward packets based on multiple header fields, such as source address, protocol type, or application port numbers. The destination-based forwarding can be thought of as *one-dimensional* packet classification.**

While several efficient solutions are known for the one-dimensional IP lookup problem, the multi-dimensional packet classification has proved to be far more difficult. While an $O(\log w)$ time scheme is known for the IP lookup, Srinivisan et al. [1] show a lower bound of $\Omega(w^{k-1})$ for k-dimensional filter lookup, where w is the number of bits in a header field. In particular, this lower bound precludes the possibility of a binary search like scheme even for 2-dimensional filters (say, IP source and destination pairs).

In this paper, we examine this lower bound more closely, and discover that the lower bound depends crucially on *conflicts* in the filter database. We then show that for two-dimensional conflict-free filters, a binary search scheme does work! Our lookup scheme requires $O(\log^2 w)$ hashes in the *worst-case*, and uses $O(n \log^2 w)$ memory. Alternatively, our algorithm can be viewed as making $O(\log w)$ calls to a prefix lookup scheme.

It has been observed in practice that filter databases have very few conflicts, and these conflicts can be removed by adding additional filters (one per conflict). Thus, our scheme may also be quite practical. Our simulation and experimental results show that the proposed scheme also performs as good as or better than existing schemes. For example, on real firewall data-sets with over 200 rules consisting of source and destination IP prefixes, our algorithm performs worst case 12 hashes. For filter sets containing *arbitrarily many* filtering rules with IP prefixes, the worst case search time guaranteed is utmost 25 hashes.

I. Introduction

Traditional routing of packets involves determination of outgoing link based on the destination address and then transferring packet data to the appropriate link interface using switching fabric. Destination-based packet forwarding treats all packets going to the same destination address identically. However important emerging applications, such as Virtual Private Networks (VPN), demand better service differentiation. Packet classification based on selected fields from packet headers provides a general mechanism to achieve this goal. Packet classification involves selection of header fields from packets, such as source and destination addresses, source and destination port numbers, protocol or even parts of URL; and then finding out the best *packet classification rule* (also called filtering rule or filter) to determine action to be taken on the packet. Since it is possible to peek at header fields corresponding to Layer 4 or above in the OSI architecture, and perform a classification lookup that uses a combination of these fields, this is commonly referred to as L4+ switching. Each packet classification rule consists of a prefix (or range of values) for each possible header field, which matches a subset of packets. As an example, consider an ISP that wants to support bandwidth guarantees for VPNs. Packet classification rules for this application can be of the type *(source network prefix, destination network prefix, guaranteed bandwidth)*. The most specific rule for a packet determines the VPN that the packet belongs to and associated bandwidth guarantee. This framework can be used in various settings. Some prominent applications include: packet filtering in firewalls, flow aggregation for MPLS tunneling, QoS routing, flow-preserving load balanced switching.

Packet classification using ad hoc mechanisms like linear search through all filtering rules is too slow in practice and a significant source of bottleneck. Hence the problem has received some attention in last 2 years. In particular, the tuple space framework proposed by Srinivasan et.al. [1] and associated simulation results suggest significant reduction in search space, while keeping memory requirement almost linear. The tuple space is formed by distinct combinations of prefix lengths (w) in the filter set. For filters containing IP prefixes, maximum prefix length for fields is $w = 32$. The number of distinct prefix length combinations is hence significantly smaller than total number of filters. However, as the number of fields k on which lookups are performed increases, size of the tuple space can grow upto $O(w^k)$. Moreover, Srinivasan et.al. [1] show that $\Omega(w^{k-1})$ hashes per lookup might be

Cisco Systems.
Computer Science, UC Santa Barbara, CA 93106
Computer Science and Engineering, UC San Diego, CA 92040

necessary in the worst case. In particular, for classification on 2 fields (2-dimensions), they prove that $2w - 1$ hashes are necessary in the worst case and sufficient. This result implies that it is not possible to perform *binary search* on hash lengths, as done for IP prefixes by Waldvogel et.al. [2], for filters with more than one field. Cross-producting scheme proposed by Srinivasan et.al. [3] suffers from a memory blowup which can be as bad as $O(n^k)$ even for very simple and natural filter sets. Gupta et.al. [4] propose an algorithm geared towards hardware implementation which suffers from a similar memory blowup. Thus existing schemes for packet classification either have bad worst case lookup times, or suffer from memory explosion. Moreover there is evidence to suggest that the time-space tradeoff for general packet classification problem is hard to bridge.

In this paper, we consider fast lookup schemes for two-dimensional filters. Since two-dimensional filters are the simplest generalization of the 1 dimensional IP lookup problem, they provide a natural setting in which to examine the limits and implications of the lower bounds proved by Srinivasan et al. [1] for the packet classification problem. Since these lower bounds imply that we cannot hope to get really fast packet classification algorithms (without exponential memory) for arbitrary filter sets, it is important to identify important practical cases where provably fast lookup algorithms can be achieved.

Srinivisan et al. show that if w is the prefix length for each of the two fields, then in the worst-case $2w - 1$ hash probes must be made by any algorithm that finds the best matching filter. When we examined this lower bound closely, we discovered that their argument depends critically on having conflicts in the filter set. (Briefly, we say that two filters conflict if they both match a packet header, but neither filter is contained in the other.) Conflicts in filters lead to distinct classification regions. While in the worst-case, n filters with k fields can have n^k conflicts, it has been observed that in practice number of conflicts are extremely small. For instance, Gupta et al. [4] report that filter databases of sizes upto 1734 had only 2581 conflicts, whereas the worst-case bound would have been $\approx 10^{13}$.

When a filter database does have conflicts, there is an elegant way to remove them by inserting additional filters covering the region of overlap. (See Hari et al. [5].) Thus, from a practical standpoint, we can assume that real databases are conflict free.

Our main contribution in this paper is to show that binary search can be used for packet classification in 2D filters *if the filters are conflict-free*. Thus we are able to identify and solve an important case of packet classification where the lower bounds do not apply. In particular, given a set of n two-dimensional filters, where each field has maximum prefix length w, our scheme has worst-case search cost $O(\log^2 w)$ and memory $O(n \log^2 w)$. Alternatively, one can view our algorithm has performing $\log w$ prefix lookups, each of which can be done in $O(\log w)$ hashes. For example, for source and destination IP prefix fields, the worst case lookup time is 25 hashes. Practically, however, number of distinct prefix lengths in filter sets tends to be much smaller. Hence the algorithm performs much better. We utilize the observation that practical filter sets are mostly conflict-free, to design an algorithm that has very good worst case lookup time which also does not suffer from any memory explosion.

The case of two-dimensional filters, while more restrictive than general filters, is important for many practical reasons. For example, application like VPN and flow aggregation for MPLS, that require use of source network *and* destination network prefixes use 2-field filters. Also, practical firewall databases contain very few distinct protocol ranges. So it is possible to breakup a firewall filter set on more than 2 fields into multiple independent 2-field filter sets without increasing memory requirement significantly.

This paper is organized as follows. Section II defines the packet classification problem formally, and introduces notation and relevant assumptions. Related work is reviewed in Section III. Important ideas related to tuple space search are reviewed in Section IV. In Section V, the conflict-free filter search algorithm is presented and analyzed. Experimental results are discussed in Section VII and the paper concludes in Section VIII.

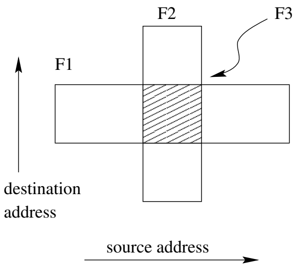
II. Problem Statement

Packet classification involves looking at k fields from each packet header. Let $P[i]$ denote the i-th field from packet P. Examples of possible fields include source address, destination address, source and destination port numbers, protocol etc. A filtering rule consists of k prefixes, one for each field from packet header. (An important exception is that ranges are often used for port values. However any range can be converted into a small set of prefixes. For example, a range $0 - 80$ can be converted into the set of prefixes that correspond to subranges $\{0 - 63, 64 - 79, 80 - 80\}$. Srinivasan et.al. [1] suggest a more efficient method that works for nested ranges.) Let F denote a filter, and let $F[i], 1 \leq i \leq k$, be the i-th prefix in the filter. Each field of a filter rule is applied to the corresponding field selected from packet header. A filter matches i-th field of a packet if prefix $F[i]$ matches $P[i]$. A filter rule is said to be a *matching filter* for packet P if it matches all fields of the packet header.

As an example, consider a filter containing two prefixes $(144.16.111.*, 128.252.*)$. These prefixes could be applied to source and destination address fields from packets. Then, the filter matches a packet with source $144.16.111.2$ and destination $128.252.169.100$; on the other hand it does not match a packet from source $144.16.111.2$ and destination $127.0.0.1$.

A packet classification database (or filter set) $\mathbf{F} = \{F_1, \ldots, F_n\}$ is a set of k-field filters. Since it is possible that more than one filter in $\mathbf{F}$ may match a given packet, we have to define what constitutes the *best matching filter*. Let F_1 and F_2 be two matching filters for packet P. If prefix $F_1[i]$ is longer than $F_2[i]$ for some i, then F_1 is said to be more specific filter in field i and F_2 is said to be less specific in field i than F_1. The best matching filter for a packet P is defined as a matching filter that is not less specific than any other matching filter. Geometrically, a filter can be considered as a set of header values that it matches. So a 2-field filter can be viewed as a rectangle containing these values. Figure 1 shows an example of the geometric view.

There might exist a pair of filters such that one filter is more specific than the other in one field and less specific in another field. In such a case, it is not possible to designate any one filter as the best matching filter. Hence such a pair of filter is said to be *conflicting*.



Filter	Source Prefix	Destination
F1	10*	0101*
F2	1011*	01*
F3	1011*	0101*

Fig. 1. Two conflicting filters.

As an example consider the filters F_1 and F_2 as shown in Figure 1. A packet with source field 10110 and destination 01010 is matched by both F_1 and F_2. Filter F_1 is more specific in the destination field while as filter F_2 is more specific in source field. In case of filter conflict, there is *ambiguity* regarding action corresponding to which filter should be taken for the packet. As proposed by Hari et.al. [5], a general way to resolve conflicts is to introduce *conflict resolution filters*. As shown in Figure 1, introduction of a new filter $F_3 = (1011*, 0101*)$ allows determination of a unique best matching filter for any possible source,destination values.

The 2-field conflict-free packet classification problem can be defined as follows. Given a conflict free filter set $\mathbf{F}$ of 2-field filters, and a packet with header fields $P[\ ]$, determine the best matching filter $F \in \mathbf{F}$.

III. Related Work

Recently several algorithms for packet classification have appeared in the literature. Many of these algorithms which provide fast lookup performance, require $O(n^k)$ memory in the worst case. In comparison, the algorithm presented here requires only $O(n \log^2 w)$ memory and still performs lookup in $O(\log^2 w)$ time for conflict free filters with 2 fields.

Linear Search and Caching The simplest approach to packet classification is to perform a linear search through all the filters. This requires $O(n)$ memory, but also takes $O(n)$ lookup time, which can be unacceptably large even for modest size filter sets.

Caching is a technique often employed at either hardware or software level to improve performance of linear search. If packets from the same flow have identical headers, packet headers and corresponding classification solution can be cached. However, performance of caching is critically dependent on having large number of packets in each flow. Also, if number of simultaneous flows becomes larger than cache size, performance degrades sharply. Note that average lookup time is adversely affected by even a small miss rate due to very high cost of linear search. Hence caching is much more useful when combined with a good classification algorithm that has a low miss penalty.

Hardware-based Solutions Large degree of parallelism can be implemented in hardware to gain speed advantage. Particularly, ternary Content Addressable Memories (CAMs) can be used very effectively for filter lookup. However, it is difficult to manufacture CAMs with wide enough words to contain all bits in a filter. CAMs with particular word width cannot be used when flexibility in filter specification to accommodate larger filters is desired. Also, large size CAMs that can accommodate, say, $16K$ filters are not yet available.

An interesting approach that relies on very wide memory accesses is presented by Lakshman et.al. [6]. The scheme computes the best matching prefix for each of the k fields of the filter set. For each filter a pre-computed n-bit bitmap is maintained. The algorithm reads nk bits from memory, corresponding to the best matching prefixes in each field and takes their intersection to find the set of matching filters. Memory requirement for this scheme is $O(n^2)$ and it requires reading nk bits from memory.

These hardware-oriented schemes rely on heavy parallelism, and represent significant hardware cost. Flexibility and scalability of hardware solutions is very limited.

Grid of Tries and Cross-producting Specifically for the case of 2-field filters, Srinivasan et.al. [3] present a trie-based algorithm. This algorithm has memory requirement $O(nw)$ and requires $2w - 1$ memory accesses per filter lookup. Also presented in [3] is a general mechanism called cross-producting which involves performing best matching prefix lookups on individual fields, and using a pre-computed table for combining results of individual prefix lookups. However, this scheme suffers from a $O(n^k)$ memory blowup for k-field filters, including $k = 2$ field filters.
Recursive-flow Classification Gupta et.al. [4] have presented an algorithm for packet classification, which can be considered a generalization of cross-producting. After best matching prefix lookup has been performed, recursive-flow classification algorithm performs cross-producting in a hierarchical manner. Thus k best matching prefix lookups and $k-1$ additional memory accesses are required per filter lookup. Though this is expected to provide significant improvement on an average, in the worst case it requires $O(n^k)$ memory. Also, for the case of 2-field filters, this scheme is the same as cross-producting and hence has memory requirement of $O(n^2)$.
Tuple Space Search Srinivasan et.al. [1] have presented an algorithm requiring $2w - 1$ hash probes per lookup on 2-field filters. They also describe a heuristic called 'tuple space pruning' which performs best matching prefix lookups on individual fields to eliminate prefix length combinations that cannot match the query. This heuristic is expected to reduce search space on an average, but does not provide any improvement in the worst case.
Binary Search on Prefix Lengths For the one-dimensional IP lookup problem, the binary search on prefix lengths scheme of Waldvogel et.al. [2] achieves $O(\log w)$ worst-case bound. It is tempting to think that one can generalize this scheme to multidimensional filters, but as the lower bound in Srinivasan et.al. [1] proves, binary search in the tuple space cannot work. In this paper, we revisit this lower bound and show that a binary search scheme is possible when the filters are two-dimensional conflict free filters.

Other schemes for the filter lookup problem have been proposed in the literature. Gupta et al. [7] proposed a heuristic HICuts that makes hierarchical cuts in the search space. It is difficult to characterize conditions under which such heuristics perform well, and the worst case memory utilization for the HICuts scheme may explode.

IV. Tuple Space Search

The basic idea behind tuple space is motivated by the observation that while filter databases contain many different prefixes or ranges, the number of distinct *prefix lengths* tends to be small. Thus, the number of distinct combinations of prefix lengths is also small. For instance, backbone routers have about $50K$ destination address prefixes, but there are only 32 distinct prefix lengths. Thus, we can divide all the prefixes into 32 groups, one for each length. Since all prefixes in a group have the same length, we can use the prefix bit string as a hash key to lookup whether a prefix exists in the database or not. This basic idea leads to a simple IP lookup scheme, which requires $O(w)$ hash lookups, *independent of the number of prefixes*. The algorithm of Waldvogel performs a binary search over the w length groups, and achieves $O(\log w)$ worst-case search time.

The tuple space idea generalizes this to multi-dimensional filters. We can define a tuple for each combination of field length, and call the resulting set *tuple space* (denoted by **T**. Since each tuple has a known set of bits in each field, by concatenating these bits in order we can create a hash key, which can then be used to map filters of that tuple into a hash table. As an example, the two-dimensional filters $F = (101*, 1*)$ and $G = (110*, 0*)$ will both map to tuple $(3, 1)$. When searching the tuple $(3, 1)$, we construct a hash key by concatenating 3 bits of the source field with 1 bit of the destination field. Thus, we can find the best matching filter by probing each tuple in turn, and keeping track of the best matching filter. (A simple, but key, observation is that either there is no match in a tuple, or there is a *unique* match.) Since the number of tuples is generally much smaller than the number of filters, even a linear search of the tuple space results in a significant improvement over linear search over the filters.

A. Pre-computation and Markers in Tuple Space

A linear search of the two-dimensional tuple space, though simple, can require $\Omega(w^2)$ hashes in the worst-case, which is too costly. A natural question is whether the optimal filter can be found without the linear search. Srinivasan et al. [1] found an algorithm, called Rectangle Search, which always finds the best matching filters in a $w \times w$ tuple space using at most $2w - 1$ hashes in the worst-case. Rectangle Search builds on two keys ideas: *pre-computation* and *markers*. The idea behind pre-computation is this: consider a tuple (i, j), where $0 \leq i, j \leq w$. A filter F in this tuple has i bits of source prefix and j bits of destination prefix. All filters that are mapped to any tuple in the *top-left quadrant* (shown by lightly shaded area in Figure 2) of (i, j) are less specific in both the fields. That is, if (i', j') is a tuple in the top-left quadrant of (i, j), then $i' \leq i$ and $j' \leq j$. Thus, we can pre-compute and store with F the best matching filter from all the tuples in the top-left quadrant of (i, j).

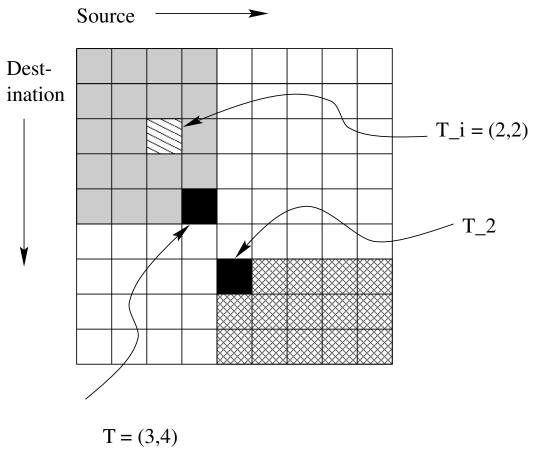


Fig. 2. Markers and Pre-computation

For instance, consider a filter $(101*, 0110*)$ which maps to the tuple $(i, j) = (3, 4)$. For all packet headers matched by this filter, we can look at filters from the top-left quadrant and determine the best matching filter *among filters that map to tuples in top-left quadrant*. For example, the best matching filter could be $(10*, 01*)$ which maps to the tuple $(i', j') = (2, 2)$.

The main advantage of pre-computation is that if we probe the tuple (i, j) with the packet header P, and get a match, then we need not search the top-left quadrant of (i, j) — the pre-computation has already determined the best filter from that set.

The second idea of markers deals with the case when there is no match on a hash probe. Each filter in a tuple (i, j) leaves a marker in all the tuples in the top-left quadrant of (i, j). The marker of a filter F is obtained by taking prefixes of its two fields. For instance, the $(2, 1)$ marker of a filter $(11001*, 0011*)$ will be $(11*, 0*)$. As shown in Figure 2, all filters in the *bottom right* quadrant (shown by cross-shaded area) of a tuple T_2 can leave markers at tuple T_2. Hence, if a hash probe of tuple T_2 fails, it is certain that there are no matching filters belonging to tuples in the bottom right quadrant.

Now, the Rectangle Search algorithm works as follows: given a packet header P, we start by probing the tuple at the bottom-left of the tuple space. If we get a match, we move one column to the right; by pre-computation, any filter from a tuple above the current tuple has been stored with the marker or filter found by the match. If there is no match, we move one row up; by the marker rule, there cannot be any filter matching to the right of the current tuple — otherwise that filter's marker in the current tuple will also have matched. Thus, each hash probe either eliminates a row or a column. Altogether after at most $2w - 1$ hashes, we find the best matching filter.

B. Lower Bound and Impossibility of Binary Search

While Rectangle Search improves upon the linear tuple space search, it falls far short of the efficiency of binary search. Recall that for the one-dimensional case, Waldvogel et.al. are able to perform a binary search on the tuple space of size w, and get $O(\log w)$ worst-case bound. Can a similar performance be achieved for two- or higher-dimensional tuple space?

A lower bound argument shows that such a bound in general is not possible. Consider a set of filters that map to tuples along diagonals, as shown by shaded cells in Figure 3. An adversary argument which shows that it is necessary to probe *all* the $2W - 1$ tuples is as follows. Given a packet header P, a lookup algorithm makes a sequence of probes into the tuples space to determine the best matching filter. Depending on this sequence, an adversary can place filters that map to appropriate tuples.

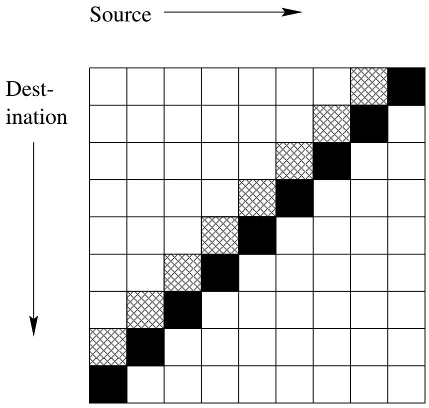


Fig. 3. Illustrating the lower bound argument for tuple space search.

For every probe that the algorithm makes on tuples along or below the main diagonal, the adversary can reply no-match by not placing a matching filter in that tuple. This will not eliminate any tuples other than the one that was probed. Also, for every probe along or above the secondary diagonal (shown by crossed-shading in figure), the adversary can construct a filter that places a matching marker in that tuple. So any such probe will not eliminate any other shaded tuple. If the algorithm does not probe some tuple from these diagonals, the adversary can create a *best* matching filter that maps to the un-probed tuple. Hence the lookup will be incorrect. Hence it is necessary to probe all the $2W - 1$ tuples in the worst case. The same lower bound can also be proved using a "decision tree model."

In this paper, we examine this lower bound more closely, and observe that if the two-dimensional filters are conflict-free, then indeed binary search performance is possible.

Note that the lower bound construction depends critically on the adversary's ability to put a filter in an un-probed tuple — but we observe that this filter will definitely conflict with existing filters.

V. Binary Search Scheme

In this section, we describe a new algorithm for 2D conflict-free filters that takes, for the first time, $O(\log^2 w)$ time per lookup *in the worst case*.

We assume that the search space is $w \times w$ tuple space. (In IPv4, the source and destination fields are 32 bits long, so $w = 32$.) We assume we have n two-dimensional conflict-free filters. These filters have been mapped to tuple space; each filter can be mapped to its unique tuple in constant time. Shortly, we will also describe how filters generate markers to be added to other tuples. In the end, all filter and markers mapped to a tuple are organized into a hash table.

Our search algorithm essentially performs a binary search over the *columns* of the tuple space, using markers and pre-computation. As shown in Figure 4, a column i in the tuple space is set of all tuples with exactly i bit source prefixes. So column i is the set of tuples $(i, j), j \leq w$.

Each filter can create markers *in the same row* into tuples from columns to its left. For example, in Figure 4 a filter mapping to tuple $(4, 3)$ can create markers in shaded tuples. Note that at this point, markers are being created only in the same row. In order to perform binary search on

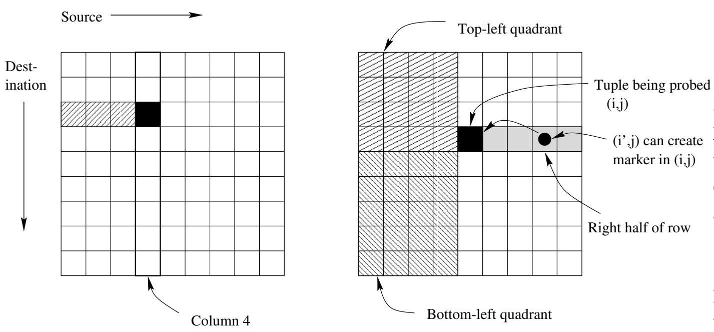


Fig. 4. Binary search on columns

columns, one half of the tuple space has to be eliminated at a time. For every hash probe into a tuple, we can divide the tuple space into 3 regions, as shown in Figure 4(b). If a matching filter or marker is found by the hash probe, then as we prove in Lemma 1, the entire left half of tuple space (i.e., top-left and bottom-left quadrants can be eliminated. On the other hand, if no matching filter or marker is found in the tuple, then there cannot be any matching filters in right half of the row. Hence, as shown in Lemma 2, if there is no matching filter in *any* tuple in the column, then entire right half of tuple space can be eliminated.

Lemma 1: If there exists a filter or marker M matching packet P in tuple (i, j), all columns $k < i$ (columns in the left half) can be eliminated from search space using *pre-computation*.

Proof: If M is a filter, then it is a better match than any possible matching filter in the top-left quadrant. Otherwise, if M is a marker, the smallest filter containing M can be pre-computed. So, if the best matching filter for packet P maps to any tuple in the top left quadrant it can be pre-computed and stored with the matching marker in tuple (i, j). Hence tuples in top-left quadrant need not be searched, and can be eliminated.
In order to eliminate tuples in the bottom-left quadrant, we use conflict-free nature of filters. Consider any filter F that placed marker M. (If M is a filter then $F = M$.) Since filters leave markers only in the same row, the filter must map to some tuple $(i', j),\ i' \geq j$. If there is any matching filter that maps to a tuple in the bottom-left quadrant, it will be less specific in first field and more specific in second field than filter F. So existence of any matching filter F' in bottom-left quadrant will imply conflict between filters F and F' and contradict the assumption that filter set **F** is conflict-free. Hence there cannot exist any matching filter mapping to tuples in bottom-left quadrant, and the all these tuples can be eliminated from search space. ■

Lemma 2: If there does not exist any marker (or filter) matching packet P in column i of tuple space, all columns $k \geq i$ (columns in the right half)can be eliminated from search space.

Proof: Suppose there exists a filter F matching packet P, which maps to some tuple (i', j) $i' \geq j$ in right half of the tuple space. Then, due to construction of markers, F will leave a marker in the tuple (i, j) (or may map to tuple (i, j)). Since F matches P, the marker (or filter F) must also match P. But there are no markers or filters matching P in the entire column, so this is a contradiction. Hence there cannot exist any matching filter that maps to a column $k \geq i$. So the entire right half of the tuple space can be eliminated. ■

From Lemma 1 and Lemma 2, the predicate "Does there exist a marker in column i matching packet P" can be used to perform binary search on columns. A simple algorithm is to probe all w tuples of the middle column i to eliminate half of tuple space. This will require $O(\log w)$ column searches, and a total number of $O(w \log w)$ hash probes. However, we can improve lookup time very significantly with the following observation.

All filters (including any marker filters) that map to tuples in column i have the same number of bits in source field. Hence concatenation of i bits of source field and the destination prefix can be considered as a single prefix. e.g., consider filters $F_1 = (1011*, 001*)$ and $F_2 = (0110*, 11001*)$. F_1 maps to tuple $(4, 3)$ and F_2 maps to tuple $(4, 5)$. In order to find any matching filter from column 4, we can take the 4 bits from field 1 and concatenate the prefix for field 2 for filters F_1 and F_2. In order to determine whether or not there exists a matching marker in column i, a *best matching prefix lookup* can be performed. A number of algorithms that solve the IP prefix lookup problem can be used for this purpose. In particular, $O(\log w)$ hashes are sufficient [2] for search within any column.

VI. ALGORITHM AND IMPROVEMENTS

The discussion so far has only said that a filter leaves markers in the same row in all columns to the left. A naive algorithm that creates markers in this manner will generate $O(nw)$ markers. Also, in order to perform best matching prefix lookup within columns, every filter and marker has to create $O(\log w)$ secondary markers in the worst case [2]. Thus a naive algorithm will have $O(nw \log w)$ total memory complexity.

However, since we perform binary search on columns, it is un-necessary to create markers in all columns. In order to reduce memory requirement, a balanced binary search tree can be created on columns. Every filter F can create markers only in columns to its left that will be be visited by the search algorithm while searching for an entry whose best matching filter is F. Since height of the balanced binary tree is $O(\log w)$, number of markers is bounded by $O(n \log w)$, and hence total memory requirement (including secondary markers) by $O(n \log^2 w)$.

Now we present details regarding construction of the data structure, and the lookup algorithm. The pseudo-code in Figure VI summarizes construction of markers for binary search on columns.

In order to perform binary search on prefix lengths within a column, any filters and markers mapped to tuples in that column will create another set of *secondary markers* [2]. Figure VI summarizes construction of secondary-markers for performing binary search on prefix lengths within a column.

The complete Binary Search on Columns algorithm for filter lookup is described in Figure VI.

Theorem 3: The *Binary Search on Columns Algorithm* finds the best matching filter in $O(\log^2 w)$ hash probes.

Proof: Height of the balanced binary search tree on columns of tuple space is utmost $\lceil \log w \rceil$. Similarly, for row-tree of any column, the balanced binary tree has utmost w nodes, and hence it is utmost of height $\lceil \log w \rceil$.

ALGORITHM MARKERS

1. Construct tuple space **T** by finding the set of tuples to which at-least one filters maps.
2. Construct balanced binary search tree on non-empty columns of **T**, called *column-tree*.
3. **for** each $F \in \mathbf{F}$ **do**
   Let T be the tuple that F maps to.
   for each ancestor T' of T in the column-tree **do**
      Insert marker into tuple T' for filter F.
      Precompute the best matching filter for newly created marker.
   end for
end for

Fig. 5. Construction of markers for Binary Search on Columns

ALGORITHM INCOLMARKERS

for each non-empty column of **T** **do**
   Construct balanced binary search tree on non-empty tuples in the column. Call this the *row-tree*.
   for every filter or marker F in the column **do**
      Let T be the tuple that F maps to.
      for each ancestor T' of T in the *row-tree*
         Insert secondary marker into tuple T' for filter F.
         Pre-compute the best matching filter/marker in *this* column for the secondary marker.
      end for
   end for
end for

Fig. 6. Construction for binary search *within* column

ALGORITHM BINARYSEARCHONCOLUMNS

node ← *column-tree*.root
best-matching-marker ← nil
repeat
 if best-in-column(node) $\neq$ nil **then**
  node ← node.right-child (columns $k > i$)
  best-marker ← best-in-column(node)
 else
  node ← node.left-child (columns $k < i$)
 end if
until node is a leaf
return pre-computed best matching filter
for best-marker

ALGORITHM BESTINCOLUMN(i)

node ← *row-tree*(column i).root
best-match ← nil
repeat
 if matching filter/marker corresponding
 to tuple at node **then**
  node ← node.right-child (columns $k > i$)
  best-match ← matching filter/marker
 else
  node ← node.left-child (columns $k < i$)
 end if
until node is a leaf
return pre-computed best matching filter in
this column for best-match

Fig. 7. Binary Search on Columns

The search algorithm traverses path root down to some leaf in the row-tree for every search within a column. Number of hashes per search within a column is hence $O(height) = O(\log w)$. Since searches within columns is equal to the longest path from root of the column-tree down to some leaf, it is equal to $O(\log w)$. Hence total time complexity of the search algorithm is $O(\log^2 w)$. ■

VII. EXPERIMENTAL RESULTS

This section describes the experimental setup and measurements we use to compare the performance of the presented algorithm with other filter lookup schemes.

A. Implementation

The Binary Search algorithm was implemented in *C++* on a UNIX machine. Main data structures used in the implementation are as follows.

- Column Tree and Row Trees: As described in previous section, one balanced binary tree was built on columns of the tuple space. Only those columns to which at-least one filter was mapped were considered. This tree is called Column Tree. For each non-empty column a separate balanced binary tree was constructed to carry out prefix lookup within that column.
- Hash Table: All filters and markers were organized in a single hash table. Hash keys were constructed using concatenation of full prefix addresses and prefix lengths.

Suitable counters were used to count total number of hash probes per classification lookup.

A.1 Empirical Results for Firewall Data-sets

Experiments were conducted using 4 industrial firewall data sets accessible to the authors. Filtering rules in each data set are of the form *(IP source prefix, IP destination prefix, source port range, destination port range, protocol)*. In order to evaluate the 2-dimensional binary search scheme, we extracted IP source and destination prefix pairs; new filters were added to resolve any conflicts. The firewall sets contained many filters of the type $(S, *)$ or $(*, D)$, i.e. with one field containing default. Such filters created a large number of conflict-resolution filters. However, as Gupta et al. [4] observe, most practical filter sets do not contain large number of conflicts.

Uniformly distributed random header fields were generated to determine average number of hash probes per lookup. Also, longest paths in column and row trees were measured to determine number of hashes per lookup in the worst case.

Table I summarizes average and worst case lookup time in terms of number hashes.

Dataset	Number of Rules	Worstcase	Average
Fwal-1	129	11	9
Fwal-2	43	6	6
Fwal-3	51	6	6
Fwal-4	143	11	9

TABLE I
LOOKUP TIME FOR FIREWALL DATA SETS

A.2 Empirical Results for Random Filter Sets

As number of filters in database grows, benefit of doing binary search become more visible. Since there do not seem to be any large filter sets available publicly, we performed experiments with random filters to ascertain scalability of the proposed algorithm. Source and destination prefixes were chosen uniformly from the MaeEast database [8]. When no filters of the type $(S, *)$ or $(*, D)$ were selected, only a negligible number of conflict-resolution filters were required. Filters with only one field specified can be separated and (IP) prefix lookup algorithms can be used for performing lookup on them. So it acceptable to generate only more complex filters for filter lookup.

In the following table, for each filter database size, number of hash probes is averaged over 10 runs. Number of distinct prefix lengths in the MaeEast prefix set is 24. As the number of filters grows, it is expected that every prefix length will get used in the filter set. So the worst case number of hash probes is expected to saturate towards the bound of $\log^2 w = 25$. This is clearly seen in Table II. The average number of hashes remains somewhat lower than the worst case. This happens because binary search tree for column search (also within column) is not full.

No. of Filters	Worst case	Average
1,000	16.0	11.3
5,000	21.2	14.7
10,000	22.0	15.4
50,000	23.7	19.8

TABLE II
LOOKUP TIME FOR RANDOM FILTERS

B. Comparison with Other Schemes

Several filter lookup schemes have been proposed in the literature. We compare these schemes for the case of 2 dimensional conflict-free filters with respect to lookup time and memory requirement. Table III compares worst case lookup time and space complexities. The Tuple Space Search algorithm of Srinivasan et.al. prunes the firewall data sets considered quite effectively. Even then the number of tuples remains almost 4 times more than the number of probes required by Binary Search. Also, in the worst case, for IP prefixes number of tuples $w^2 = 1024$ gives very large lookup time. Rectangle Search reduces worst case lookup time upto $2w - 1 = 63$ for IP prefixes. The Pruned Tuple Space scheme proposed by Srinivasan et.al. [1] performs two prefix lookups and attempts to eliminate some tuples from search. In the worst case, no tuples may get eliminated. Though in practice pruning works quite well, the additional overhead of two prefix lookups makes it perform worse than the binary search algorithm presented here. The Grid-of-tries scheme performs $2w - 1$ memory accesses per lookup. This is the same as Rectangle Search. The Cross-producting scheme of Srinivasan et.al. [3] performs only two prefix lookups per filter lookup. The penalty for such efficient lookup time is however severe—$O(n^2)$ memory requirement, which becomes prohibitive even for databases of modest sizes. The Recursive Flow Classifier scheme of Gupta et.al. [4] is identical to cross-producting, for 2 dimensional filters. As can be seen from the above table, binary search on columns provides significantly better time complexity, without consuming large amount of memory.

Scheme	Lookup time	Memory usage
Tuple space search	$O(w^2)$	$O(n)$
Rectangle search	$O(w)$	$O(nw)$
Pruned tuple space	$O(w^2)$	$O(n)$
Grid of tries	$O(w)$	$O(nw)$
Cross-producting	$O(\log w)$	$O(n^2)$
This Algorithm	$O(\log^2 w)$	$O(n \log^2 w)$

TABLE III
COMPARISON OF WORST CASE LOOKUP TIME AND SPACE COMPLEXITIES

Experimental results indicate that the constants involved in the proposed algorithm are quite small, and the algorithm is competitive against other schemes *in average case also*. Srinivasan et.al. provide lookup time for the same firewall data sets that were used in our experiments. For the Tuple Space Search algorithm, lookup time is simply the number of distinct tuples to which at-least one filter is mapped. Their Pruned Tuple Space scheme reduces search space almost to the same number of tuples probed by Binary Search on Columns. However, with the expense of two additional IP lookups (which take $\log W = 5$ hashes per lookup), their lookup time almost doubles. We do not have performance figures for the same data sets for other schemes. However, for Grid of Tries scheme, lookup time is simply time for first prefix lookup, followed by $W - 1$ memory accesses for second prefix. Also, for the Cross-producting scheme, lookup time is always equal to 2 independent prefix lookups. Table IV compares worst case lookup time on the firewall data sets for these schemes.

Scheme	Fwal-1	Fwal-2	Fwal-3
Tuple Space Search	41	41	12
Pruned tuple space	21	17	15
Grid of Tries	36	36	36
Cross-producting	10	10	10
This Algorithm	11	6	6

TABLE IV
COMPARISON OF WORST CASE LOOKUP TIME ON FIREWALL DATA-SETS

VIII. CONCLUSIONS

Performing packet classification based on multiple fields at high speed while maintaining low memory requirement is a hard problem. It is possible to performing classification lookups on 1 field very efficiently by doing binary search on prefix lengths [2]. A theoretical lower bound of $\Omega(w^{k-1})$ for arbitrary filters [1] indicates that it is important to recognize practical special cases of filter sets. We show that it is possible to perform classification very efficiently on conflict-free filters. We have presented an algorithm that performs lookups in $O(\log^2 w)$ time on 2-field conflict free filters. This is the fastest $2D$ filter lookup algorithm with small memory costs known to the authors. When the fields under consideration are IP prefixes, this translates into *worst case* lookup time of 25 hashes. The average case performance of this algorithm significantly better than other algorithms in literature for 2 dimensional conflict free filter. The proposed algorithm also has very good space complexity of $O(n \log^2 w)$. The proposed algorithm is scalable to large filter sets and can be implemented very easily in software. Filter lookup is a difficult problem, and as shown in this work adding the conflict-free constraint makes it feasible to solve the problem very efficiently. Conflict-free filters is a practically feasible constraint. It remains an interesting open question whether this (or similar) constraints can be used to improve performance and provide better worst case bounds for other schemes.

REFERENCES

[1] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *Proceedings of SIGCOMM'99*, 1999.

[2] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed ip routing lookups," in *Proceedings of SIGCOMM'97*, 1997.

[3] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," in *Proceedings of SIGCOMM'98*, 1998.

[4] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proceedings of SIGCOMM'99*, 1999.

[5] A. Hari, S. Suri, and G. Parulkar, "Detecting and resolving packet filter conflicts," in *Proceedings of IEEE INFOCOMM'2000*, 2000.

[6] T.V. Lakshman and D. Stidialis, "High speed policy-based packer forwarding using efficient multi-dimensional range matching," in *Proceedings of SIGCOMM'98*, 1998.

[7] Pankaj Gupta and Nick McKeown, "Packet classification using hierarchical intelligent cuttings," in *Hot Interconnects VII*, August 1999.

[8] Merit Inc., "Ipma statistics," http://nic.merit.edu/ipma.