Available online at www.sciencedirect.com

SCIENCE DIRECT®

Computer Networks 50 (2006) 1406–1423

Computer Networks

www.elsevier.com/locate/comnet

Packet classification using diagonal-based tuple space search ☆

Fu-Yuan Lee *, Shiuhpyng Shieh

Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan 300, Taiwan

Received 8 January 2004; received in revised form 27 March 2005; accepted 21 June 2005
Available online 16 August 2005

Responsible Editor: D. Stiliadis

Abstract

Multidimensional packet classification has attracted considerable research interests in the past few years due to the increasing demand on policy based packet forwarding and security services. These network services typically involve determining the action to take on packets according to a set of rules. As the number of rules increases, time for determining the best matched rule for an incoming IP packet will increase and subsequently incur long processing delay. To address this issue, in this paper we propose a two-dimensional packet classification algorithm which focuses on reducing time for classification while keeping reasonable memory requirement in practice. Our approach extends the tuple space framework and then allows performing binary search on the tuple space. To our knowledge, the proposed scheme is the first binary search scheme on two-dimensional tuples. With the proposed scheme, given a filter set with n two-dimensional filters, it requires only O(log(w)) hash operations to determine the best matched filter, where w is the maximum prefix length of filters. The proposed scheme achieves fast packet classification, and according to our experimental results, it does not require huge memory space. This makes it useful for network applications that require high speed packet classification.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Computer networks; Network security; High speed network; Layer 4 switching

1. Introduction

Many network services require packet classification, such as packet filtering for VPN and firewall, and packet forwarding for QoS routing. These services typically involve classification of

☆ This work is supported in part by National Science Council and Institute for Information Industry.
* Corresponding author. Tel.: +886 3 571212.
E-mail address: leefy@csie.nctu.edu.tw (F.-Y. Lee).

incoming packets so as to determine subsequent processing for each packet. This classification is achieved with *filters* applied to each incoming packet. Each filter consists of prefixes of packet header fields, and specifies an action to take on the packets matching all the prefix specifications. (Note that range format is usually used to specify port numbers. However, any range of values can be efficiently converted into a union of a set of prefixes [18].) Upon arrival of an incoming packet, packet classification is first performed to determine an appropriate filter for the packet. Subsequently the action specified by the filter is performed.

In this service paradigm, it is important to accelerate packet classification since it can subsequently reduce the processing delay of each packet. Linear search through all the filters is usually too slow in practice. Hence, issues and techniques for accelerating packet classification has been intensively investigated in the last few years [6,9,14] For the simplest case of packet classification, i.e. each filter only specifies one prefix, Waldvogel et al. [21] proposed a scheme performing binary search on the prefix lengths of filters. In their approach, filters are grouped according to prefix lengths. As a result, filters in the same group have the same prefix lengths and thus can be searched in one hash operation. Moreover, since these groups can be sorted according to the prefix lengths, binary search can be applied on the set of groups. Consequently, only $O(\log(w))$ hash operations are required to find the longest matched prefix for an incoming IP packet, where w represents the maximum prefix length.

Srinivasan et al. [16] proposed the *tuple space framework* which adopts the concept of searching on prefix lengths to cope with *multidimensional packet classification*. In multidimensional packet classification, each filter defines prefix specifications on multiple packet header fields, and therefore each filter has more than one prefix length. The vector of prefix lengths of a filter is called a *tuple*, and the *tuple space* is the set of distinct tuples in a filter set. Similar to Waldvogel's work [21], filters mapped to the same tuple can be searched using one hash operation. Since the number of tuples is generally much smaller than the number of filters, this approach can significantly reduce the search space. Consequently, linear search through all of the tuples is faster than that through all filters. However, as the number of fields k used for classification increases, the total number of tuples can grow up to $O(w^k)$. In this case, though the search space is reduced, linear search through all the tuples may still has long delay.

Based on the basic tuple space search, two other algorithms, namely Rectangle Search [16] and Binary Search on Columns [22], which both further improve the search efficiency in the tuple space are proposed. Both algorithms focus on two-dimensional packet classification. Given n filters, Rectangle Search requires $2 * w - 1$ hashes per lookup. The memory space requirement is $O(n * w)$. As shown in [16], without using more memory space, it is impossible to obtain a packet classification algorithm running faster than Rectangle Search. Warkhede et al. [22] re-examined this claim and discovered that Srinivasan's argument heavily depends on having conflicts in the filter set. However, it has been shown that conflicts can be removed by inserting new filters into the original filter set [3,10]. Thus, by assuming the filter sets are conflict free, the search efficiency can be further improved. Using Warkhede's approach, it requires $O(n * \log^2(w))$ memory space, and uses only $O(\log^2(w))$ hashes to determine the best matched filter.

In addition to the tuple-space based approaches, there are other schemes for multidimensional packet classification proposed in the literature. Grid-of-Tries [18] is a trie-based algorithm for two-dimensional packet classification. It requires $O(n * w)$ memory space and $2 * w - 1$ memory accesses per lookup. Cross producting [18] requires $d * w$ memory access and $O(n^d)$ memory space, where d represents the number of dimensions. Baboescu et al. [2] proposed Extended Grid-of-Trie (EGT) algorithm to cope with general multidimensional packet classification. Similar to the original Grid-of-Trie approach, EGT requires $O(n * w)$ memory space and $O(w)$ memory accesses. Recursive-flow classification (RFC) [7] is a general multidimensional packet classification scheme which can determine the best matched filter in constant time. Although the search efficiency is high, RFC suffers from memory blowup. Similar to RFC, other schemes such as Hi-Cuts [8], Segment Tree [19], and Range-Matching [13], all suffer from the same memory blowup problem if the number of filters becomes large. Thus, as mentioned above, existing packet classification algorithms either suffer from bad search performance or require huge memory space.

Particularly in this paper, we are concerned with the design of fast two-dimensional packet classification algorithms, which are considered important for many emerging source-address involved packet forwarding services/applications [12], such as multicast [5,15,20], the measurement of traffic between networks, and some resource reservation protocols. In these services/applications, packet forwarding decisions are made according to a set of rules, each of which contains specifications for both source and destination addresses of an IP packet to match. An Internet router providing these services has to accelerate its packet classification by employing fast two-dimensional packet classification algorithms so as to keep pace with the increasing high volume of network traffic.

To address this issue, in this paper, a fast two-dimensional packet classification algorithm based on tuple space search is proposed. In the original construction of tuple space, binary search cannot work well, and this motivates a new construction of tuple space which is suitable for binary search on tuples. The proposed tuple space construction is similar to original construction presented in [16]. The use of pre-computation and markers were originally proposed in [16], and they appear in the proposed scheme as well. The major difference between our tuple space construction and the original one is on the introduction of a new auxiliary filter, called *resolver*. As we shall see in Section 2.2, with the original construction, the set of remaining tuples for a successful probe into a tuple (i.e. a matched filter is found at the tuple) *overlaps* the remaining tuples for a failing probe. Therefore, binary search fails to operate in the original tuple space construction. While, in our scheme, with the aid of resolvers, the set of remaining tuples for a successful probe and that for a failing probe are *disjoint*. In other words, our approach can divide a tuple space into two disjoint parts no matter a probe is successful or failing, and this characteristic makes binary search on the tuple space effective.

The efficiency in search time costs a larger storage requirement. In the proposed scheme, given n two-dimensional filters, where each prefix is at most w bits, time complexity for searching is $O(\log(w))$ and space complexity is $O(n^2)$ in the worst case. The worst case may occur if the filters severely conflict with each other. Fortunately, as reported [7], the number of conflicts in practice is much smaller than in the worst case. In other words, the worst case unlikely happens in practice. The contribution of this paper is to show that binary search on tuple space is possible, and our scheme is practical for network applications which require high speed packet classification.

This paper is organized as follows. In Sections 2 and 3, fundamentals of tuple space search and basic ideas behind the proposed scheme are described. In Section 4, proposed diagonal-based tuple space search algorithm is presented. Evaluation and comparison are discussed in Section 5. Finally, a brief conclusion is given in Section 6.

2. Fundamentals of tuple space search

In this section, packet classification problem is formally defined, the basic idea of tuple space search is reviewed, and the idea behind the proposed algorithm is described.

2.1. Problem statement

A *classifier* is a set of filters and each filter is composed of prefix specifications on one or more selected packet header fields. A filter f consisting of k prefix specifications is often referred to as a k-dimensional filter. A k-dimensional filter can be represented as $(f[1], f[2], \ldots, f[k])$, where each $f[i]$ is a prefix specification on a packet header field. A packet p is said to match a filter f if and only if prefixes of the selected packet header fields of p are correspondingly the same as the prefixes specified by f. Since it is possible that a packet can match more than one filter and each filter may specify different actions, it is necessary to determine which action to take. Generally, each filter is associated with a *priority*. Among the filters that a packet p matches, the filter with the highest priority is selected as the best matched filter. Given a classifier containing n filters and a packet p, packet classification is the process of determining the best matched filter for the packet p. This paper is concerned about the two-dimensional packet classification problem. We assume that each filter is two-dimensional, and prefixes are expressed as a bit string ending with the wildcard symbol. For instance, "10*" specifies the most two significant bits are "10" and the "*" denotes the wildcard symbol.

2.2. Fundamentals of the tuple space search

Tuple space search is motivated by two observations. First, the number of distinct combinations of prefix lengths is usually much smaller than the number of filters in a classifier. For instance in the case of destination-based packet forwarding, although each router can have hundreds to thousands of prefixes in the routing table, the number of distinct prefix lengths is at most 32. For a two-dimensional classifier, where each filter specifies the prefixes of source addresses and destination addresses, it can have at most 1024 ($=32 * 32$) distinct combinations of prefix lengths. In the following context, each distinct combination of prefix length is called a *tuple*. The *length vector* of a tuple refers to the combination of prefix lengths. For example, (24,16) is a tuple which indicates that filters mapped to this tuple have 24-bit prefix specification in its first dimension and 16-bit prefix specification in the second dimension.

The second observation is that search on filters mapped to the same tuple requires only one hash operation. Since filters mapped to the same tuple have the same number of bits in each field correspondingly, the concatenation of prefixes of each filter can be used to create a hash key. The hash keys are then used to map filters in the same tuple to a hash table. Specifically, each tuple has a hash table used to store filters mapped to the tuple. Consider a two-dimensional filter $f = (x, y)$ mapped to a tuple T. Let $x\|y$ denote the concatenation of x and y, and $H(.)$ be the hash function used to create hash keys of filters. If $v = H(x\|y)$, then filter f is stored in the v-th entry in tuple T's hash table. To test if a packet p can match any filter in a tuple T, a hash key is created by concatenating the required number of bits from the selected packet header fields according to the length vector of T. Then, filters indexed by the hash key of p are then compared to the packet sequentially. If the packet matches one of the filters indexed by the hash value, a matched filter is found. In this way, the tuple space framework can significantly reduce the search space. Even without any additional improvements, linear search through all the tuples is generally faster than linear search through all filters.

Some readers might be interested in the way to create hash functions which are used to generate hash keys in tuple space. One simple approach is to use w^2 hash functions, each of which is especially associated with a tuple in the tuple space. For instance, a hash function which takes a 32-bit input is used to create hash keys for filters mapped to tuple (16, 16), and another hash function taking a 33-bit input is used for filters in tuple (16, 17). Furthermore, perfect hash functions [11] can be used to minimized hash collisions. However, this approach would incur hidden costs in creating and maintaining the w^2 hash functions. One way to eliminate the cost, for instance, is to use only one hash function, which takes a $2w$-bit input. In this case, to generate a hash key for a filter (x, y), we first need to append a required number of 0 s or 1 s to the end of $x\|y$ (in order to make it a $2w$-bit input), and then use the resulting $2w$ bits as the input to the hash function. Currently, there have been several studies on creating hash functions that can produce hash keys for filters in an appropriate way, such as semi-perfect hash functions [17], and hashing using multiple hash functions [4]. In this paper, we assume that $H(\cdot)$ represents a hash function which can automatically append its input to $2 * w$ bits and create hash keys for filters.

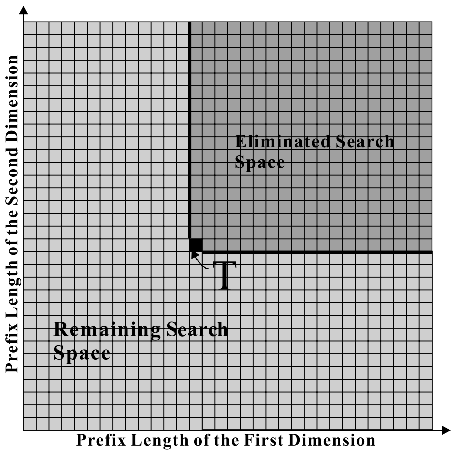
Search on tuple space can be further improved based on two ideas, namely *pre-computation* and *markers*. To describe the two ideas, several notations must be introduced first. Given a two-dimensional tuple $T_a = (i, j)$ where i, j denote the number of bits in the first and the second dimensions respectively, the rest of tuples can be partitioned into three disjointed sets, $S(T_a)$, $L(T_a)$ and $IC(T_a)$. A tuple $T_b = (m,n)$, where $T_b \neq T_a$, is an element of $S(T_a)$ if $m \leqslant i$, $n \leqslant j$. Similarly, T_b is an element of $L(T_a)$ if $m \geqslant i$, $n \geqslant j$. If T_b is neither in $S(T_a)$ nor in $L(T_a)$, then T_b is in $IC(T_a)$. Fig. 1 shows the partition of a two-dimensional tuple space into three sets.

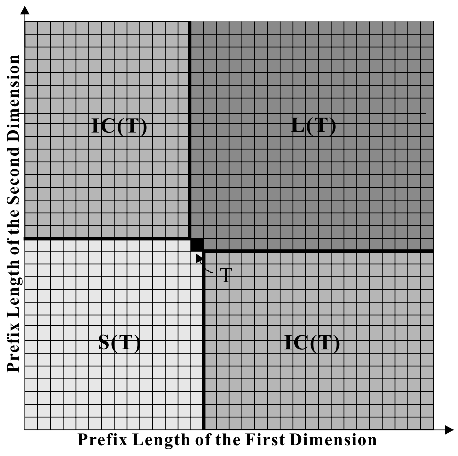
Since the length vector of each tuple in $L(T)$ is coordinate-wise greater than T, a filter f mapped to a tuple in $L(T)$ can leave a marker in T. The marker is a filter obtained by using only $T[i]$ bits of the i-th field of f, where $T[i]$ represents the i-th element of T's length vector.

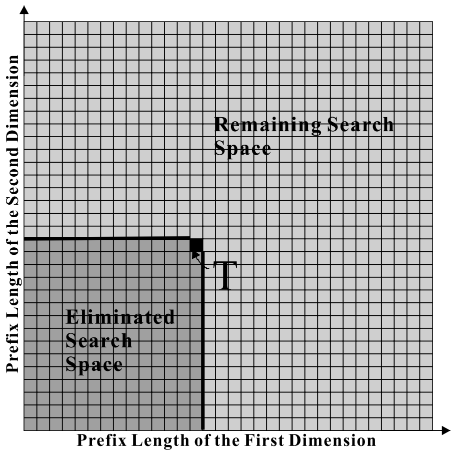
Similarly, since the length vector of each tuple in $S(T)$ is coordinate-wise smaller than T, for each filter, say f in T, it is possible to pre-compute the best matched filter of f in the set of filters that are mapped to a tuple in $S(T)$ and store it with the filter.

Consider a tuple space in which pre-computation is completed, and each filter leaves markers in tuples belonging to $S(T)$ where T denotes the tuple the filter maps into. Then, if no matched filter is found in a tuple T for a given packet, filters mapped to tuples in $L(T)$ can be eliminated from the search space. This is because if there exists a matched filter mapped to the tuple in $L(T)$, its marker entry in T will have a match. Thus, if the probe in a tuple T fails, the search space can be restricted to the filters mapped to the tuples in $S(T)$ and $IC(T)$, as shown in Fig. 2.

Similarly, if the probe in T obtains a matched filter, then filters mapped to tuples in $S(T)$ can be eliminated from the search space. This is because if another matched filter mapped to a tuple in $S(T)$, it has been pre-computed and stored with the matched filter in T. In other words, if the probe in T returns a match, the search space can be restricted to the filters mapped to the tuples in $L(T)$ and $IC(T)$, as shown in Fig. 3.


Eliminated Search Space
T
Remaining Search Space
Prefix Length of the Second Dimension
Prefix Length of the First Dimension
Fig. 2. Partition of the tuple space if the probe in tuple T fails.


IC(T)
L(T)
T
S(T)
IC(T)
Prefix Length of the Second Dimension
Prefix Length of the First Dimension
Fig. 1. Partition of the tuple space.


Remaining Search Space
T
Eliminated Search Space
Prefix Length of the Second Dimension
Prefix Length of the First Dimension
Fig. 3. Partition of the tuple space if the probe in tuple T succeeds.

2.3. Proposed tuple space search strategy

Based on the basic idea of tuple space search presented above, a new tuple space construction is proposed to support binary search on tuples. In the new tuple space construction, $IC(T)$ and $S(T)$ can be eliminated from the search space if a matched filter is found in T. The idea behind the new construction is based on the following observation. Given a packet p, consider that a matched filter f found in tuple T. For any filter g mapped to a tuple in $IC(T)$, g can be a matched filter of p if and only if g and f are *overlapped*, that is, $f[i]$ is a prefix of $g[i]$, and $g[j]$ is a prefix of $f[j]$. In other words, any filter mapped to a tuple in $IC(T)$ can be eliminated from the search space if it does not overlap with f.

Next, consider those filters which overlap with the matched filter f. Filters which overlap with f can be eliminated with the aid of auxiliary filters called *resolvers*. Consider a filter g mapped to a tuple in $IC(T)$ overlaps with f, a resolver r is created by taking the longer prefixes in each dimension from f and g. It is clear that r is mapped to a tuple in $L(T)$. In addition, the best matched filter (f or g) can be pre-computed and stored with r after the pre-computation process. In this way, filter g can be eliminated from search space. Consequently, by using resolvers, filters mapped $IC(T)$ can be eliminated whenever a matched filter is found in T.

Consider a classifier F, we say that the tuple space of F is *filter conflict resolved* if and only if one of the following criteria are satisfied: (1) each filter in F does not overlap with any other filters in F. That is, for any pair of (f_i,f_j), $f_i \neq f_j$, f_i does not overlap with f_j. (2) for each pair of overlapped filters (f_i,f_j), $f_i \neq f_j$, there must be a filter which is equivalent to the resolver o f_i and f_j. Similarly, we say that a tuple space is *filter-marker conflict resolved* if the following criterion is satisfied: for any pair of filter and marker (f_i,m_j), where f_i denotes a filter in F, and m_j represents a marker of a filter f_j, if f_i overlaps with m_j then there must be a filter equivalent to the resolver of f_i and m_j. (Note that so far, we do not specify how the markers are generated. It is also worthy to notice that a filter-marker conflict resolved tuple space is filter conflict resolved. However, a filter conflict resolved tuple space is not necessary to be filter-marker conflict resolved.)

For instance, consider a small classifier containing two filters: $f_1 = (10^*, 100111^*)$ and $f_2 = (101^*, 10000^*)$. Since f_1 does not overlap with f_2, this classifier is filter conflict resolved. However, it is not filter-marker conflict resolved because f_1 may be overlapped with f_2's markers, e.g. (101*, 100*). So far, we have not described the way markers are generated, and this is just an example to illustrate the way of constructing a filter-marker conflict resolved tuple space. To make the tuple space filter-marker conflict resolved, the conflicts between f_1 and f_2's markers must be resolved. Similarly, the conflicts between f_2 and f_1's markers must be resolved as well. It is not hard to find that the number of resolvers created depends on the way markers are created. Later in this section, we will present the details of marker creation. After all the resolvers are generated, by definition, the resulting tuple space is filter-marker conflict free. Finally, for each filter f, including the filters in F, markers and resolvers, mapped to a tuple T, its best matched filter information (computed from $S(T)$) is then pre-computed and stored with f.

Then, given a filter-marker conflict resolved tuple space, as proved in Lemma 1, if there is a matched filter in tuple T, $IC(T)$ and $S(T)$ can be eliminated from search space.

Lemma 1. *Given a filter-marker conflict resolved tuple space, if there is a filter or marker in tuple T which can match a given packet p, then filters mapped to tuples in S(T) and IC(T) can be eliminated from the search space.*

Proof. In short, $S(T)$ is eliminated by pre-computation and $IC(T)$ is eliminated by resolvers. Since any matched filter in $S(T)$ can be pre-computed and stored with f, it is clearly that filters mapped to tuples in $S(T)$ can be eliminated from search space.

Next, consider the filters which are mapped to tuples in $IC(T)$. If filter h mapped to a tuple in $IC(T)$ and is the best matched filter for packet P, then h overlaps with f. Afterwards, since the tuple space is filter-marker conflict resolved, there must be a filter which is equivalent to the resolver generated by both f and h, and the resolver is mapped to a tuple in $L(T)$. The resolver is certainly a better matched filter than f and h. Since the best matched filter information is stored with the resolver, h can also be eliminated from the search space.

If h is not a matched filter for the given packet p, filter h can thus be eliminated from search space. Therefore, in summary it is unnecessary to search filters mapped to tuples in $IC(T)$ no matter whether there is matched filter mapped to a tuple in $IC(T)$ or not. □

3. Diagonal-based tuple space search

Based on Lemma 1, a new tuple space search algorithm is proposed. The proposed scheme, called *diagonal-based tuple space search algorithm*, applies binary search on tuples and thus requires only O(log(w)) hashes to determine the best matched filter in a two dimensional filter set. Assume the search space is a $w * w$ square tuple space, and the filter set F contains n two-dimensional filters. Each field of a filter is a string of bits (at most w bits) representing the prefix of a packet header field. In addition to the filters in F, markers and resolvers are used to create a filter-marker conflict free tuple space. Details for generating markers as well as resolvers, and the proposed packet classification algorithm are presented next.

First, we describe how markers are created. Given a filter set F with n two-dimensional filters, consider a filter f mapped to tuple (i, j). Let $s = \min(i, j)$. Then markers of filter f are created and inserted into the set of tuples which are in the line from (i, j) to (s, s) and from (s, s) to $(1, 1)$. For example, as shown in Fig. 4, filters mapped to tuple (16, 24) leave markers in tuples (16, 23), (16, 22), (16, 21),...,(16, 17), (16, 16), (15, 15),...,(2, 2), (1, 1).

After markers of all filters in F are created, then resolvers are created. Reflecting to the way that markers are created, creating resolvers is quite easy. It consists of two steps. First, resolvers are created for each pair of overlapped filters in the original filter set F. After that, the markers of these resolvers are generated. Next, in the second step, a resolver is created if there is a filter overlapped with a marker in a diagonal tuple. Consider a filter mapped to a tuple $T(i, j)$ and, without lose of generality, let $i < j$. In this step, only conflicts need be examined between this filter and markers in diagonal tuples from $(i + 1, i + 1)$ to $(j - 1, j - 1)$, shown in Fig. 5. Afterwards, resolvers created leave their markers in the tuple space. Finally, pre-computation is performed for all the filters (including the filters in the original classifier F, markers and resolvers). In this way, a filter-marker conflict resolved tuple space is constructed.

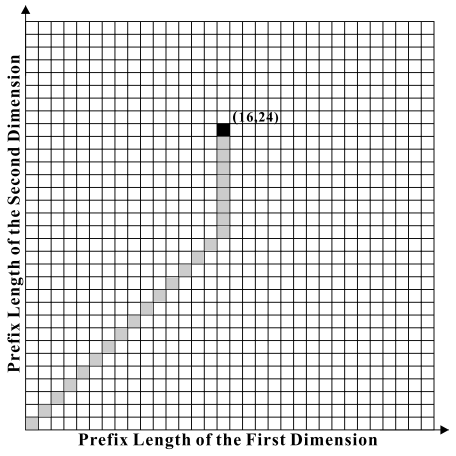

(16,24)
Prefix Length of the Second Dimension
Prefix Length of the First Dimension

Fig. 4. Generating markers in the tuple space.

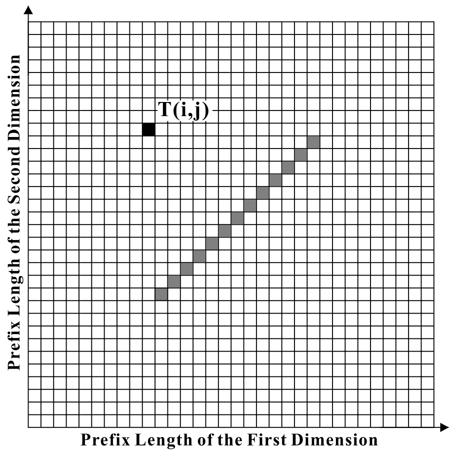
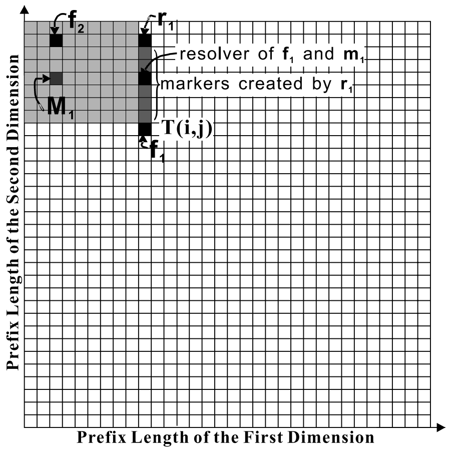

T(i,j)
Prefix Length of the Second Dimension
Prefix Length of the First Dimension

Fig. 5. The diagonal tuples examined in order to resolve filter-markers conflicts for filters mapped to tuple $T(i, j)$.

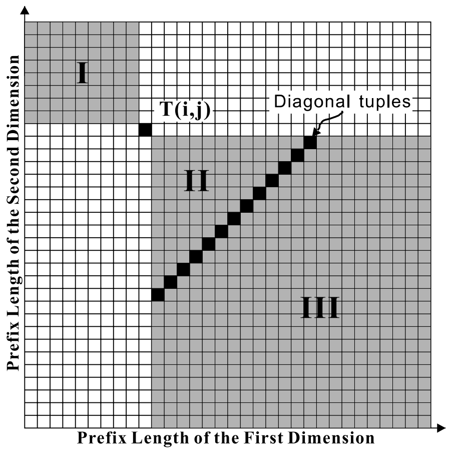
It is worthy to note that the first step is to construct a filter conflict resolved tuple space, and the second step is to resolve conflicts between filters and markers. We will show that in the second step, for a filter f_1 mapped to a tuple $T(i,j)$, where $i < j$, resolving the conflicts between the filter f_1 and markers mapped to diagonal tuples from $(i+1, i+1)$ to $(j-1, j-1)$ is sufficient for constructing a filter-marker conflict resolved tuple space.

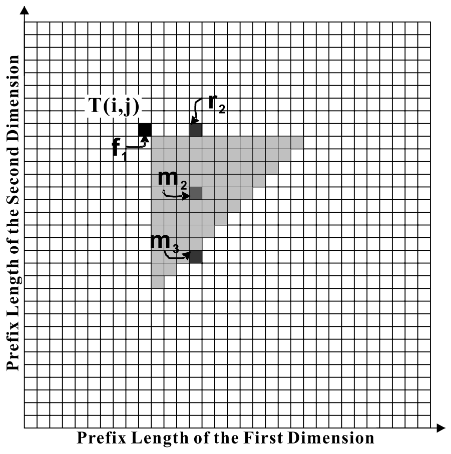
Notice that a marker can be overlapped with a filter f_1 mapped to a tuple $T(i,j)$ only if the marker is mapped to a tuple in $IC(T)$. In other words, for a filter f_1 mapped to a tuple $T(i,j)$, we only have to consider markers mapped to tuples in $IC(T)$ and resolve possible conflicts between f_1 and the markers. To resolve these conflicts, as shown in Fig. 6, we first partition the $IC(T)$ into four areas: areas I, II, III and diagonal tuples.

Next, we discuss the way to resolve possible conflicts between f_1 and markers in each area. First, consider the case that there is a marker m_1 in area I overlapped with f_1. As shown in Fig. 7, there must exist a filter f_2 which generates m_1. f_2 is mapped to a tuple in area I, and is overlapped with f_1. (m_1 is assumed to be overlapped with f_1. m_1 and f_2 have the same prefix bit string in their first dimension, and the bit string of m_1's second dimension is a prefix of the bit string of f_2's second dimension. Since m_1 is overlapped with f_1, f_2 is overlapped with f_1.) Therefore, there must have been a resolver r_1 created from f_1 and f_2. Recall that r_1 also generates its markers in the tuple space. We can find that one of these markers will certainly be identical to the resolver of f_1 and m_1. As a result, resolving the conflict between f_1 and m_1 is redundant and can be reduced. By the same argument, we do not need to examine and resolve the conflicts between f_1 and markers in area I.


f2
r1
resolver of f1 and m1
markers created by r1
M1
T(i,j)
f1
Prefix Length of the Second Dimension
Prefix Length of the First Dimension
Fig. 7. Resolving the conflicts between filter f and markers in area I.

Second, consider the case that there is a marker m_2 in area II, and m_2 is overlapped with f_1. As shown in Fig. 8, the existence of m_2 implies that there must be a marker m_3 in the diagonal tuple


I
T(i,j)
Diagonal tuples
II
III
Prefix Length of the Second Dimension
Prefix Length of the First Dimension
Fig. 6. Consider a tuple $T(i,j)$. The IC(T) can be partitioned into four areas: area I, II, III, and diagonal tuples.


T(i,j)
r2
f1
m2
m3
Prefix Length of the Second Dimension
Prefix Length of the First Dimension
Fig. 8. Resolving the conflicts between filter f and markers in area II.

(in the same column with m_2) such that m_3 and m_2 have the same prefix bit string in the first dimension, and the prefix bit string in the second dimension of m_3 is actually a prefix of the bits strings in the second dimension of m_2. Since m_2 is assumed to be overlapped with f_1, m_3 is overlapped with f_1 as well. Thus, a resolver, r_2, will be created from m_3 and f_1. Moreover, we can also find that r_3 actually is identical to the resolver created from m_2 and f_1. Since the two resolvers are identical, we can skip resolving the conflict between f_1 and m_2. Similarly, by the same argument, we do not need to resolve the conflicts between f_1 and markers in area II.

Finally, consider the case that there is a marker m_4 in area III and m_4 is overlapped with f_1. The existence of m_4 implies that there must be a filter f_3 generating m_4 in area III. Since m_4 is overlapped with f_1, we can know that f_3 is also overlapped with f_1. Therefore there must be a resolver r_3 created from f_1 and f_3 in the tuple space. r_3 generates its markers in two different ways reflecting to its position in the tuple space. Let (x, y) denote the tuple that r_3 mapped into. Then, if $x < y$, r_3 is above the diagonal of the tuple space, and r_3 leaves its markers in tuples $(x, y-1)$, $(x, y-2), \ldots,$ (x, x), $(x-1, x-1), \ldots, (1, 1)$. If $x > y$, r_3 is below the diagonal, and it leaves markers in tuples $(x-1, y)$, $(x-2, y), \ldots, (y, y)$, $(y-1, y-1), \ldots,$ $(1, 1)$. If $x = y$, r_3 is mapped to a diagonal tuple, and its markers will be in tuples $(x-1, y-1)$, $(x-2, y-2), \ldots, (1, 1)$.

As shown in Fig. 9, when r_3 is above the diagonal, there will be resolvers created from f_1 and the markers, generated by r_3, in diagonal tuples. Notice that one of these resolvers is identical to the resolver created from f_1 and m_4. Thus, we can know that resolving the conflict between f_1 and m_4 is unnecessary because the corresponding resolver has been created. If r_3 is mapped to a diagonal tuple, we can find the same result. Next, consider the case that r_3 is below the diagonal. As shown in Figs. 10 and 11, we can also find that the resolver created from f_1 and m_4 is identical to one of the horizontal markers generated by r_3, or it will be identical to one resolver created from f_1 and the markers in diagonal tuple, which is generated by r_3.

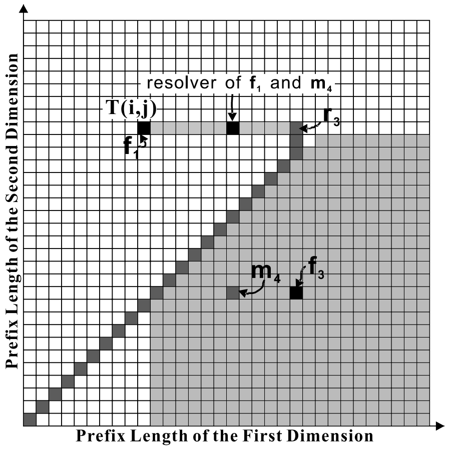

resolver of f1 and m4
T(i,j)
r3
f1
m4
f3
Prefix Length of the Second Dimension
Prefix Length of the First Dimension

Fig. 9. Resolving the conflicts between filter f_1 and markers in area III. The resolver of f_1 and f_3 is above the diagonal.

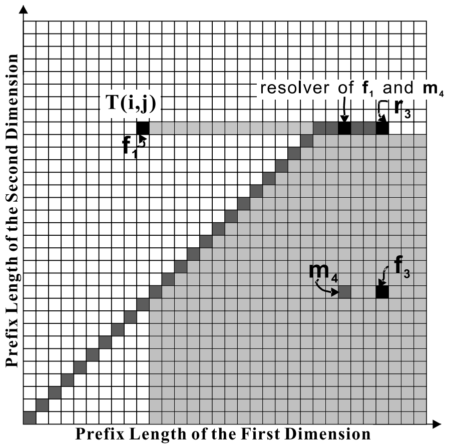

resolver of f1 and m4
T(i,j)
r3
f1
m4
f3
Prefix Length of the Second Dimension
Prefix Length of the First Dimension

Fig. 10. Resolving the conflicts between filter f and markers in area III. The resolver of f_1 and f_3 is below the diagonal. This figure shows that the resolver of f_1 and marker m_4 is identical to a marker of r_3.

Based on previous discussion, we show that the proposed approach can efficiently construct a filter-marker conflict resolved tuple space. Here, we give an example of constructing an filter-marker conflict resolved tuple space. Consider a classifier containing two filters $f_1 = (10*, 100111*)$ and $f_2 = (101*, 10000*)$. f_1 leaves following markers: (10*, 10011*), (10*, 1001*), (10*, 100*), (10*, 10*), (1*, 1*), and f_2 leaves following markers: (101*, 1000*), (101*, 100*), (10*, 10*), (1*, 1*).

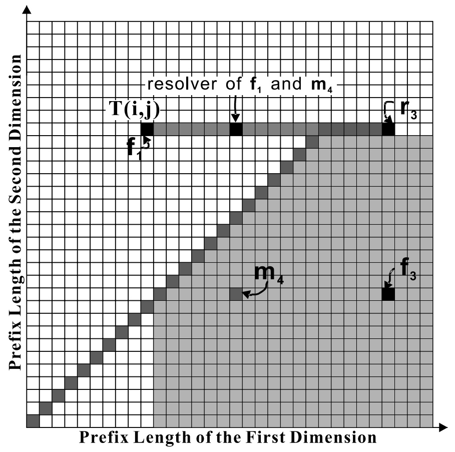

resolver of f1 and m4
T(i,j)
r3
f1
m4
f3
Prefix Length of the Second Dimension
Prefix Length of the First Dimension

Fig. 11. Resolving the conflicts between filter f and markers in area III. The resolver of f_1 and f_3 is below the diagonal. This figure shows that the resolver of f_1 and marker m_4 is identical to a resolver created from f_1 and a marker of r_3.

Since f_1 is not overlapped with f_2, the tuple space is filter conflict resolved. Next, We can find that f_1 only overlaps with f_2's marker (101*, 100*) and that f_2 does not overlap with f_1's markers. In this example, only one resolver is created, i.e. (101*, 100111*), and it leaves following markers: (101*, 10011*), (101*, 1001*), (101*, 100*), (10*, 10*), (1*, 1*). In this tuple space, we can find that filters may leave identical markers, and the filter-marker conflict resolved tuple space has two original filters f_1 and f_2, one resolver, and nine markers.

Consider the problem of finding the best matched filter in a given filter set F. The first step is to construct a filter-marker conflict free tuple space. Then, by every hash probe into a diagonal tuple T, the tuple space is divided into two regions, as shown in the Fig. 12. If a matched filter is found in a diagonal tuple, then as proved in Lemma 1, region 2 can be eliminated from the search space. On the other hand, if no matched filter is found, then there cannot be any matched filters in $L(T)$. In other words, Region 1 can be eliminated from the search space.

Thus, it is clear that if there is a matched filter found in a diagonal tuple $T = (m, m)$, and in the same time, if there is no matched filter found in tuple $(m + 1, m + 1)$, then the remaining search space can be restricted to the set of tuples:

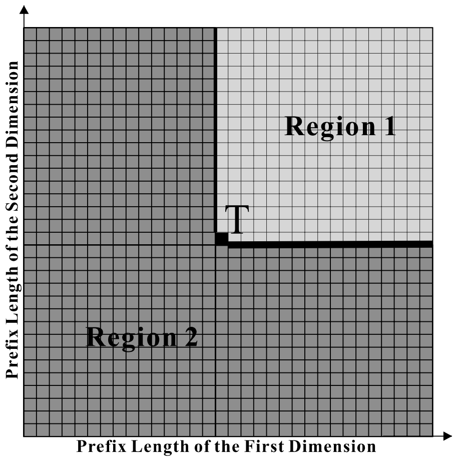

Region 1
T
Region 2
Prefix Length of the Second Dimension
Prefix Length of the First Dimension

Fig. 12. Probing a diagonal tuple would divide the tuple space into to regions.

$\{(i, j)\,|\,i = m, m \leqslant j \leqslant w \ \text{ or } \ j = m, m \leqslant i \leqslant w\}$. In this case, the tuple T is called *the last matched diagonal tuple*, denoted as T_{lmdt}. For instance, as shown in Fig. 13, if $T_{lmdt} = (16, 16)$, then the remaining search space includes tuples: (32, 16), (31, 16), (30, 16)...,(16, 16), (16, 17),...,(16, 31), (16, 32).

In the proposed tuple space search algorithm, the first step is to find T_{lmdt} which helps reduce the search space drastically. Notice that given two diagonal tuples $T_i = (i, i)$ and $T_j = (j, j)$, T_j is either in $L(T_i)$ or in $S(T_i)$. In other words, every pair of

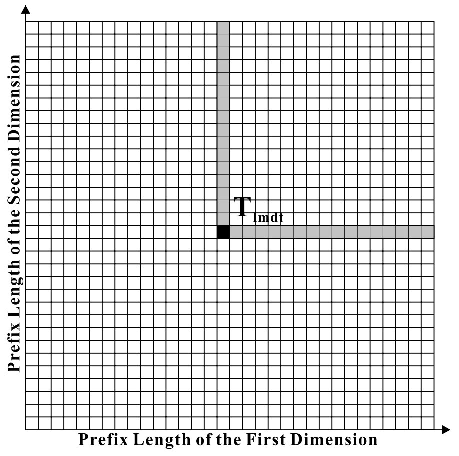

Tlmdt
Prefix Length of the Second Dimension
Prefix Length of the First Dimension

Fig. 13. Example for $T_{lmdt} = (16, 16)$.

diagonal tuples are comparable. Then, with markers and pre-computation, T_{lmdt} can be found by binary search on diagonal tuples. This is because if a matched filter is found in a diagonal tuple T_i, the search space is then restricted in $L(T_i)$. On the other hand, if there is no matched filter found in T_i, the search space is restricted in $S(T_i)$. As a result, since there are only w diagonal tuples, it requires O(log(w)) hashes to determine T_{lmdt}.

Once T_{lmdt} is determined, a naive search algorithm is to probe all remaining tuples and thus requires O(w) hashes. However, the search time can be further reduced by applying binary search on the remaining tuple. Thus, only O(log(w)) hash probes in total is required. Our improvement is based on observation that all the remaining tuples are either in the same column or row with T_{lmdt}. Similar to the case of determining T_{lmdt} in the set of diagonal tuples, with markers and pre-computation, tuples in the same column (or row) can be considered a sorted list of objects and thus binary search can be applied. In this way, only two more binary search, where one for tuples in the same column and the other for tuples in the same row, are sufficient to determine possible matched filters. In summary, it requires at most $3 * \log(w)$ hashes to find the best matched filter in F for a given packet p.

4. Tuple space construction and search algorithm

So far, the basic idea of the proposed tuple space construction and the skeleton of the diagonal-based binary search algorithm is presented. In this section, the construction of tuple space is further improved such that the number of markers and resolvers are reduced, while in the same time, retaining the same search efficiency. In this section, we described details of the enhanced tuple space construction and the proposed diagonal-based binary search algorithm.

As mentioned previously, a filter f mapped to tuple (i,j) leaves $(max(i,j) - 1)$ markers. Consequently the classifier in total creates $O(n * w)$ markers. To reduce the number of markers, the functionality of markers is examined. In the proposed search strategy, markers are used fundamentally to guide the search algorithm to find f if the best matched filter is f. In other words, a filter f can only generate markers in the tuples that will be probed by the search algorithm while searching for the best matched filter is f. Consequently, only $O(n * \log(w))$ markers are created. In comparison with $O(n * w)$, the number of markers is significantly reduced.

4.1. Tuple space construction

To describe the tuple space construction algorithm, several notations must be defined. First, a tuple is a *non-empty tuple* if there is at least one filter, marker or resolver mapped to the tuple. Then all of the non-empty tuples in the tuple space can be partitioned into a set of *tuple groups*. A tuple group, denoted as TG(i), is the collection of non-empty tuples that are in the same column or row with a diagonal tuple $T = (i, i)$ and that are mapped to $L(T)$. The set of non-empty tuples of TG(i) in the same column is denoted as TG(i).col. Similarly, the set of non-empty tuples of TG(i) in the same row is denoted as TG(i).row. For example, $\text{TG}(10).\text{col} = \{(i, 10) |$ if $(i, 10)$ is a non-empty tuple and $11 \leqslant i \leqslant w\}$ and $\text{TG}(10).\text{row} = \{(10, i) |$ if $(10, i)$ is a non-empty tuple and $11 \leqslant i \leqslant w\}$. $\text{TG}(10) = \text{TG}(10).\text{col} \cup \text{TG}(10).\text{row}$.

Algorithm 1. Construct balanced binary search trees for tuple groups

```
1:  for $i = 1$ to $w$ do
2:      if TG($i$).col $\neq \emptyset$ then
3:          Construct a balanced binary search tree on TG($i$).col, denoted
            as TG($i$).col-tree
4:      end if
5:      if TG($i$).row $\neq \emptyset$ then
6:          Construct a balanced binary search tree on TG($i$).row, denoted
            as TG($i$).row-tree.
7:      end if
8:  end for
9:  Construct a balanced binary search tree on non-empty diagonal tuples.
    denoted as diagonal-tuple-tree.
```

Pseudo-code in Algorithm 2 shows the construction of a filter-marker conflict resolved tuple space. From lines 2–5, all the filters create markers in diagonal tuples. Afterwards, resolvers are created by the pseudo code in Algorithm 3. Next, balanced binary search trees are created for each tuple groups and diagonal tuples. This is accomplished by the pseudo code shown in Algorithm 1. At this step, for each tuple group, say TG(i), two balanced binary search trees, denoted as TG(i).row-tree and TG(i).col-tree for non-empty tuples in the tuple group are constructed, where the former denotes the tree constructed using tuples in the same row while the

Algorithm 2. The construction of a filter-marker conflict resolved tuple space

```
1: Construct the tuple space.
2: Create resolvers for each pair of overlapped filters in $F$
3: for all filter $f$ (including filters in $F$ and the resolvers created previously) do
4:    Let $T = (i, j)$ be the tuple that $f$ mapped to, and $s = \min(i, j)$
5:    filter $f$ leaves a marker in the diagonal tuple $(s, s)$.
6: end for
7: Construct diagonal-tuple-tree using Algorithm 1
8: for all filter $f$ (including filters in $F$ and resolvers) do
9:    Let $f$ mapped to tuple $T$ that is a element of TG($i$).
10:   for each ancestor tuple $T'$ of tuple $(i, i)$ in diagonal-tuple-tree do
11:      if $T' \in S(T)$ then
12:         Insert a marker into tuple $T'$ for filter $f$.
13:      end if
14:   end for
15: end for
16: Resolve conflicts between filters and markers in the diagonal tuples using Algorithm 3
17: Construct row-trees and col-trees for each tuple group, using Algorithm 1.
18: for all filter $f$ (including filters in $F$ and resolvers) do
19:   Let $f$ mapped to tuple $T$ that is a element of TG($i$).
20:   if $T$ is a element of TG($i$).row then
21:      foreach ancestor node $T'$ of $T$ in TG($i$).row-tree do
22:         if $T' \in S(T)$ then
23:            Insert a marker into tuple $T'$ for filter $f$.
24:         end if
25:      end for
26:   else if $T$ is a element of TG($i$).col then
27:      for each ancestor tuple $T'$ of $T$ in TG($i$).col-tree do
28:         if $T' \in S(T)$ then
29:            Insert a marker into tuple $T'$ for filter $f$.
30:         end if
31:      end for
32:   end if
33: end for
34: Pre-computation using Algorithm 4.
```

Algorithm 3. The creation of resolvers that resolve conflicts between filters and markers

```
1: for all filter $f$ in $F$ do
2:    for $i = 1$ to $w$ do
3:       for all filter or marker $g$ in tuple $(i, i)$ do
4:          if $f$ is overlapped with $g$ then
5:             Create a resolver for $f$ and $g$, and insert the resolver into the tuple space.
6:          end if
7:       end for
8:    end for
9: end for
```

Algorithm 4. The pre-computation of the best matched filter for each filter in the tuple space

```
1: for all Tuple $T \in$ tuple space do
2:    for all filter or marker $f \in T$ do
3:       for all Tuple $T' \in S(T)$ do
4:          for all filter $g \in T'$ do
5:             if $f$ match $g$ then
6:                Set $g$ as the best matched filter of $f$.
7:             end if
8:          end for
9:       end for
```

```
10:     end for
11:   end for
```

latter denotes the tree constructed using tuples in the same column. Moreover, a balanced binary search tree, denoted as *diagonal-tuple-tree*, is constructed which is built on the non-empty diagonal tuples.

From lines 8–28, the rest of markers are created and inserted to the tuple space. Consider a filter f in tuple T and without loss of generality assume that tuple T belongs to TG(i).col of a tuple group TG(i). Filter f first leaves markers in the tuples that are in $S(T)$ and that are on the path from tuple T to the root tuple in the balanced binary search tree TG(i).col-tree. Then, f leaves markers in the tuples of $S(T_d)$ that are on the path from tuple T_d to the root tuple in the *diagonal-tuple-tree*, where T_d denotes the diagonal tuple (i, i). Note that consider two tuples T_a and T_b in the same binary search tree, tuple T_b is in the right subtree of tuple T_a if T_b is in $L(T_a)$. On the other hand, T_b is in the left subtree if T_b is in $S(T_a)$. At the final step, pre-computation is performed and its pseudo-code is shown in Algorithm 4.

4.2. Binary search scheme

The diagonal-based tuple space search algorithm is described in Algorithm 5. First, the algorithm performs binary search to determines T_{lmdt} using balanced binary search tree, *diagonal-tuple-tree*. Next, the search algorithm traverses the row tree and column tree corresponding to the T_{lmdt}. In this way, a best matched filter can be determined. Theorem 1 shows that O(log(w)) hash

Algorithm 5. The proposed binary search algorithm

```
1:  best-matching-filter ← nil
2:  Tuple T ← diagonal-tuple-tree.root
3:  repeat
4:     if a matching filter or marker f found
         at Tuple T then
5:        T_lmdt ← T
6:        T ← T.right-child
7:        if f is a filter then
8:           best-matching-filter ← f
9:        else
10:          best-matching-filter ← The
               pre-computed best matching
               filter stored with marker f.
11:       end if
12:    else
13:       T ← T.left-child
14:    end if
15: until T is a leaf node in the
      diagonal-tuple-tree
16: Let T be represented as (i, i)
17: Let T ← TG(i).col-tree.root
18: repeat
19:    if matching a filter or marker
         at Tuple T then
20:       T ← T.right-child
21:       if f is a filter then
22:          best-matching-filter ← f
23:       else
24:          best-matching-filter ← The
               pre-computed best matching
               filter stored with marker f.
25:       end if
26:    else
27:       T ← T.left-child
28:    end if
29: until T is a leaf node in TG(i).
      col-tree.root
30: Let T ← TG(i).row-tree.root
31: repeat
32:    if matching a filter or marker
         at Tuple T then
33:       T ← T.right-child
34:       if f is a filter then
35:          best-matching-filter ← f
36:       else
37:          best-matching-filter ← The
               pre-computed best matching
               filter stored with marker f.
38:       end if
39:    else
40:       T ← T.left-child
41:    end if
42: until T is a leaf node in TG(i).col-row.root
43: Output best-matching-filter.
```

operation is sufficient to determine the best matched filter.

Theorem 1. *The binary search algorithm finds the best matched filter in* O(*log*(w)) *hashes.*

Proof. The search algorithm traverses path from root down to some leaf in the diagonal-tuple-tree and subsequently traverses two binary search trees associated with the diagonal tuple T_{lmdt}. Height of the balanced binary search tree on diagonal tuples is at most $\lceil \log(w) \rceil$. Similarly the height of balanced binary search trees in each tuple group are also at most $\lceil \log(w) \rceil$. Therefore, the total number of hashes equals to O(log(w)), and thus time complexity of the search algorithm is O(log(w)). □

4.3. Dynamic update of filter sets

In addition to the determination of the best matched filter for a packet, some applications, such as firewall, may have the demand to dynamically insert or delete filter rules. To insert a new filter rule, as we do in constructing a filter-marker conflict resolved tuple space, the first step is to resolve possible conflicts between the new filter and original filters. Then, markers of the new filter are inserted into the tuple space, and subsequently the resolves for the new markers and original filters are created. Finally, pre-computation is performed.

Deleting a filter from a classifier is more complicated than inserting a new one. First, we have to remove all the resolvers and markers created from this filters. Notice that different filters may leave identical markers. Thus, a marker can actually be removed only after all the filters or resolver creating the marker are deleted. After the removal of resolvers and markers of the deleted filter, pre-computation is executed on the new tuple space.

It is clear that the need of pre-computation makes our scheme more geared toward static filter set or filter set that changed infrequently. It is hard to guarantee that the insertion and deletion of a filter can be completed at line-rate. However, for applications that can tolerate some delay in adjusting filter sets, our approach is applicable and provides fast packet classification.

5. Performance evaluation and comparison

This section first gives complexity comparison with other packet classification algorithms. Subsequently, the experimental setup and measurement result on the memory requirement of the proposed algorithm are described.

5.1. Complexity comparison

Table 1 shows the comparison of proposed scheme with existing classification algorithms which focus on two-dimensional packet classification. Comparison is made in terms of search time and memory space requirement. In the comparison, n denotes the number of filters in the classifier and w represents the maximum prefix length of filters.

As can be seen from Table 1, linear search through all of the tuples w^2 hashes and subsequently may incur too much delay in the worst case. Rectangle Search, Grid-of-Trie and Extended Grid-of-Trie have the same number of memory access. Binary Search on Columns provides better search efficiency while the proposed scheme has the best search performance. The penalty for the fast packet classification is the large memory space used. However, the worst case is unlikely to happen unless the filters in the classifier are severely overlapped with each other.

5.2. Estimation of memory requirement

Although the proposed scheme may use large memory space in the worst case, the memory requirement is likely to be much smaller in practice. This will be shown by some typical experiments later in this section. Since there is no

Table 1
Comparison of time and space complexities

Scheme name	Search time	Memory space
Linear tuple space search	O(w^2)	O(n)
Rectangle search	O(w)	O(nw)
Grid-of-trie	O(w)	O(nw)
Extended grid-of-trie	O(w)	O(nw)
Binary search on columns	O($\log^2(w)$)	O($n\log^2(w)$)
Proposed scheme	O($\log(w)$)	O(n^2)

publicly available large filter sets, it is hard to know the memory usage of the proposed scheme under the use of real-life filter sets. Thus, artificially created filter sets are used to estimate the memory requirement of the proposed scheme. According to the algorithm presented in Algorithm 2, there are at most $O(n * \log(w))$ markers. It is clear that the number of resolvers is the dominating factor of the memory requirement. To estimate the memory space overhead caused by resolvers, *Resolver overhead* is defined as (number of resolvers/number of filters). In the following, various types of filter sets are examined so as to identify characteristics that may affect the resolver overhead.

In our experiment, prefixes in the publicly available BGP tables [1] are used to construct filter sets. Prefixes are first categorized according to their originating AS numbers. Prefixes with the same originating AS number are classified into the same category and each category may has one to thousands of prefixes. Fig. 14 shows the distribution of number of prefixes in each category.

Given a set of prefixes, it is possible that a prefix $prefix_1$ may be the prefix of another prefix $prefix_2$. Then, we say that the *covered count* of a prefix is k if the prefix is a prefix of other k prefixes in the set. For example, consider a set of five prefixes expressed in CIDR format, *140.0.0.0/8, 140.113.0.0/16, 140.113.1.0/24, 140.113.2.0/24, 140.113.3.0/24*. Then, the covered count of *140.0.0.0/8* is 4. The average covered count is 1.4.

A two-dimensional filter set can be created by cross-producting prefixes of two categories. As mentioned before, without large publicly available

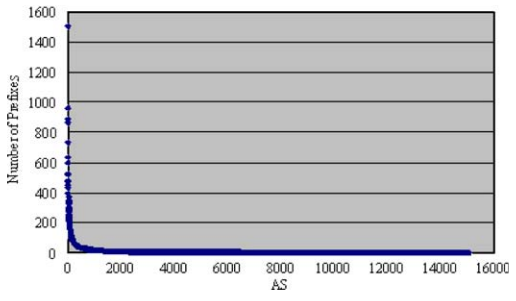


Fig. 14. Number of prefixes with the same originating AS number.

classifiers, so far the best we can do is to create random filter sets. In our experiments, the source prefix and destination prefix are randomly selected from two categories respectively. However, since most of the categories have only one prefix, cross-producting prefixes of two categories can create classifiers with a small number of filters only. To generate a large filter set, one straightforward approach is to combine categories. This would create groups of prefixes, where each group can have sufficient number of prefixes. Then, it will be much easier to create large filter sets by cross-producting prefixes of two groups. In our experiments, the average covered counts of categories are used to joint categories together. According to the average covered count, all the categories can be partitioned into five sets, as shown in Table 2.

Then, we choose to combine the top eight categories, in terms of number of prefixes, to create a larger set of prefixes. Tables 3–7 show the selected top eight categories in each group.

Afterwards, five groups of prefixes can be constructed by joining prefixes of the top eight categories. Table 8 shows the number of prefixes and average covered count of the generated prefix groups.

Now, there are 5 prefix groups, where each group has sufficient number of prefixes. By cross producting prefixes of two groups, 10 combinations can be generated. Table 9 shows the 10 types of combination and the corresponding *covered level*. The *covered level* of a two dimensional classifier is defined as the product of the average covered counts of the two groups which constitutes the classifier. Notice that covered count is defined to characterize one dimensional filter sets while covered level is for multi-dimensional classifiers.

Table 2
Number of categories in the five groups

Group ID	Range of covered count	# of categories
1	$2 \leqslant$ covered count	8
2	$1.5 \leqslant$ covered count < 2.0	27
3	$1 \leqslant$ covered count < 1.5	113
4	$0.5 \leqslant$ covered count < 1	954
5	$0 \leqslant$ covered count < 0.5	13,988

Table 3
Information of the 8 ASs in the first group

AS number	# of prefixes	Average covered count
14654	147	2.251701
9930	85	2.082353
3776	49	2.142857
4787	44	2.636364
9129	25	2.440000
4758	23	2.173913
12150	23	3.347826
4800	12	2.166667

Table 4
Information of the 8 ASs in the second group

AS number	# of prefixes	Average covered count
9583	231	1.800866
11172	171	1.508772
19864	105	1.600000
3573	71	1.647887
10029	66	1.863636
9425	52	1.576923
2706	49	1.530612
8795	36	1.555556

Table 5
Information of the 8 ASs in the third group

AS number	# of prefixes	Average covered count
5668	205	1.341463
13609	197	1.208122
3464	147	1.374150
19916	127	1.102362
15105	93	1.129032
4471	80	1.012500
8717	72	1.305556
9829	70	1.285714

Table 6
Information of the 8 ASs in the fourth group

AS number	# of prefixes	Average covered count
65529	888	0.740991
7132	861	0.739837
4323	600	0.901667
6197	518	0.532819
4355	395	0.772152
27364	290	0.948276
4755	225	0.946667
6140	224	0.696429

Table 7
Information of the 8 ASs in the fifth group

AS number	# of prefixes	Average covered count
701	1503	0.140386
1239	962	0.214137
3908	884	0.363122
702	729	0.106996
7843	632	0.370253
852	526	0.076046
6198	477	0.343816
209	472	0.271186

Table 8
Information of the five groups

Group ID	# of prefixes	Average covered count
1	408	2.311275
2	781	1.658131
3	991	1.243189
4	4001	0.765059
5	6185	0.318998

Table 9
Ten types of combinations

Combination ID	(Group ID, Group ID)	Covered level
1	(1, 2)	3.832397
2	(1, 3)	2.873352
3	(2, 3)	2.061370
4	(1, 4)	1.768262
5	(2, 4)	1.268568
6	(3, 4)	0.951113
7	(1, 5)	0.737292
8	(2, 5)	0.528940
9	(3, 5)	0.396575
10	(4, 5)	0.244052

For instance, consider combination 1 which is made from group 1 and group 2, its covered level is $3.832397 = (2.311275 * 1.658131)$.

In the following experiments, we attempt to show that the resolver overhead is related to the covered level. The observation is that classifiers created from combinations with high covered level will have higher resolver overhead. To confirm the observation, we randomly created classifiers with different number of filters for each combination. In our experiments, for each combination, classifiers consisting of 100, 200, ..., 1900, 2000, 4000, 8000, 10,000 filters respectively were created. For each size, 500 randomly created filter sets were tested. In each test, the resolver overhead was calculated. Fig. 15 shows experimental results of classifiers with the number of filters fewer than 2000. Table 10 shows the results of classifiers with 4000, 8000, 10000 filters.

It is clearly that the experimental results shown in Fig. 15 and Table 10 confirm the observation. For instance, classifiers created from combination 1, which has the highest covered level, have the highest resolver overhead. On the other hand, classifiers created from combination 10 have the lowest resolver overhead.

Additionally, the highest resolver overhead observed in our experimental result is 0.5. It indicates that, in the experiment, resolvers require half

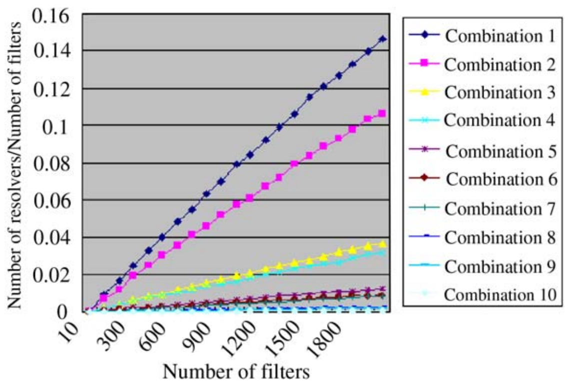


Fig. 15. Resolver overhead of classifiers generated from different combinations and with different sizes.

Table 10
Resolver overhead of classifiers of size 4000, 8000 and 10000

Combination ID	Resolver overhead in different filter set size		
	4000	8000	10000
1	0.265551	0.444161	0.520234
2	0.19495	0.335861	0.394862
3	0.068694	0.126703	0.153178
4	0.057959	0.103177	0.122906
5	0.023154	0.042083	0.050485
6	0.016713	0.031242	0.037887
7	0.015111	0.029033	0.035152
8	0.005154	0.009521	0.011782
9	0.003872	0.007244	0.008948
10	0.001429	0.002592	0.003339

memory space compared to the memory space used by original filters in the classifier, that is, if the covered level of real-life classifiers is small, our approach may not require huge memory space to store resolvers.

Finally, we use a firewall database on hand to illustrate the memory requirement of the proposed scheme. The firewall is currently deployed to protect a network with twenty personal computers, a web server, a samba server, a SMTP server, and three ftp servers. There are in total 71 firewall rules. We use the source address and destination address specifications to construct a two-dimensional classifier. The covered level of the classifier is 0.01. Only four resolvers (the resolver overhead is 0.056) and in total 162 two-dimensional filters in the tuple space are created. Memory space used by the tuple space is about 5K bytes. This evaluation result may help show that the memory requirement of the proposed scheme is feasible for network devices such as routers, firewalls or NAT devices.

6. Conclusion

Two-dimensional packet classification is important for many network applications. Many schemes have been proposed to accelerate this operation. In this paper, we proposed a binary search algorithm that performs packet classification in $O(\log(w))$ hash operations. The memory usage can reach $O(n^2)$ in the worst case. However, if the covered count of a filter set is small, the memory requirement is reasonably low. In other words, our approach can provide fast packet classification operation for classifier with small covered level. For classifiers with large covered level, our scheme is still applicable if search efficiency is much more important than storage considerations.

References

[1] Bgp table data. Available from: <http://bgp.potaroo.net/>.

[2] F. Baboescu, S. Singh, G. Varghese, Packet classification for core routers: Is there an alternative to cams? in: Proceedings of INFOCOM 2003, 2003.

[3] F. Baboescu, G. Varghese, Fast and scalable conflict detection for packet classifiers, in: Proceedings of International Conference on Network Protocols 2002, 2002.
[4] A. Broder, M. Mitzenmacher, Using multiple hash functions to improve IP lookups, in: Proceedings of INFOCOM, vol. 3, Apr. 2001, pp. 1454–1463.
[5] B. Cain, S.E. Deering, I. Kouvelas, B. Fenner, A. Thyagarajan, Internet Group Management Protocol, Version 3, Internet Engineering Task Force, RFC 3376, Oct. 2002 [Online]. Available from: <http://www.rfc-editor.org/rfc/rfc3376.txt>.
[6] A. Feldmann, S. Muthukrishnan, Tradeoffs for packet classification, in: Proceedings of IEEE INFOCOM, 2000, pp. 1193–1202.
[7] P. Gupta, N. McKeown, Packet classification on multiple fields, in: Proceedings of ACM SIGCOMM, 1999, pp. 147–160.
[8] P. Gupta, N. McKeown, Classifying packets with hierarchical intelligent cuttings, IEEE Micro 20 (1) (2000) 34–41.
[9] P. Gupta, N. McKeown, Algorithms for packet classification, IEEE Network (March/April) (2001) 24–32.
[10] A. Hari, S. Suri, G. Parulkar, Detecting and resolving packet filter conflicts, in: Proceedings of IEEE INFOCOM, 2000, pp. 1203–1211.
[11] D.E. Knuth, The Art of Computer Programming: Sorting and Searching, vol. 3, Addison-Wesley Professional, 1998.
[12] V. Kumar, T. Lakshman, D. Stiliadis, Beyond best effort: router architectures for the differentiated services of tomorrow's Internet, IEEE Communications Magazine 36 (May) (1998) 152–164.
[13] T.V. Lakshman, D. Stiliadis, High-speed policy-based packet forwarding using efficient multi-dimensional range matching, in: Proceedings of ACM SIGCOMM, 1998, pp. 203–214.
[14] C. Macian, R. Finthammer, An evaluation fo the key design criteria to achieve high update rates in packet classifiers, IEEE Network (November/December) (2001) 24–29.
[15] J. Moy, Multicast Extensions to OSPF, Internet Engineering Task Force, RFC 1584, Mar. 1994 [Online]. Available from: <http://www.rfc-editor.org/rfc/rfc1584.txt>.
[16] V. Srinivasan, S. Suri, G. Varghese, Packet classification using tuple space search, in: Proceedings of ACM SIGCOMM, 1999, pp. 135–146.
[17] V. Srinivasan, G. Varghese, Fast address lookups using controlled prefix expansion, ACM Transaction on Computer Systems 17 (1) (1999) 1–40.
[18] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel, Fast and scalable layer four switching, in: Proceedings of ACM SIGCOMM, Sep. 1998, pp. 191–202.
[19] C.-F. Su, High-speed packet classification using segment tree, in: Proceedings of IEEE GLOBECOM, 2000, pp. 582–586.
[20] D. Waitzman, C. Partridge, S.E. Deering, Distance Vector Multicast Routing Protocol, Internet Engineering Task Force, RFC 1075, Nov. 1988 [Online]. Available from: <http://www.rfc-editor.org/rfc/rfc1075.txt>.
[21] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, Scalable high speed ip routing lookups, in: Proceedings of ACM SIGCOMM, September 1997, pp. 25–36.
[22] P. Warkhede, S. Suri, G. Varghese, Fast packet classification for two-dimensional conflict-free filters, in: Proceedings of IEEE INFOCOM, 2001, pp. 1434–1443.

Fu-Yuan Lee received the BS degree in computer science from National Chiao Tung University in 1998. He is currently a Ph.D. student in the Department of Computer Science and Information Engineering at National Chiao Tung University. His research interests are in the areas of computer networks and network security.

Shiuhpyng Shieh is a professor and former chairman of Department of Computer Science and Information Engineering of National Chiao Tung University. He is also, and the president of Chinese Cryptology and Information Security Association (CCISA), which is the largest and a highly respectable academic organization on information security research in Taiwan. He has worked as advisor to many institutes, such as National Security Bureau, GSN-CERT/CC, National Information and Communication Security Task Force. Before joining NCTU, He participated in the design and implementation of the B2 Secure XENIX at IBM, Federal Sector Division, Gaithersburg, Maryland. He also designed and developed NetSphinx, a network security product, for Formosoft Inc., which is awarded 1999 network product of the year, Taiwan.

He received the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Maryland, College Park. He is a senior member of IEEE, and an editor of ACM Transactions on Information and System Security, Journal of Computer Security, and Journal of Information Science. He was on the organizing committees of numerous conferences, such as ACM conference on Computer and Communications Security, IACR Asiacrypt. Dr. Shieh published over a hundred academic articles, including papers, patents, and books. Recently he received the Outstanding Research Award from National Chiao Tung University for his academic achievement in research, and the Outstanding Achievement Award from Executive Yuan of Taiwan. His research interests include internetworking, distributed operating systems, and network security.