Available online at www.sciencedirect.com

SCIENCE DIRECT®

Computer Communications 27 (2004) 1637–1646

computer communications

www.elsevier.com/locate/comcom

Accelerating network security services with fast packet classification

Shiuhpyng Shieh[1], Fu-Yuan Lee[1,*], Ya-Wen Lin[1]

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

Received 12 September 2003; revised 15 April 2004; accepted 5 May 2004

Available online 2 June 2004

Abstract

To protect a network, secure network systems such as intrusion detection system (IDS) and firewall are often installed to control or monitor network traffic. These systems often incur substantial delay for analyzing network packets. The delay can be reduced with fast packet classification, which can effectively classify network traffic, and consequently accelerate the analysis of network packets. In the last few years, many researchers devoted to providing fast packet classification methods for multidimensional classifier. However, these methods either suffer from poor performance and huge storage requirement, or are lack of dimension scalability. In this paper, we propose a packet classification method based on tuple space search, and use the multidimensional binary search tree (Kd-tree) to improve search performance. The proposed scheme requires only $O(d \log W)$ search time and controlled storage requirement, where d is the number of dimensions, and W is the utmost bit length for specifying prefixes in a classification rule. It features fast packet classification, and supports dynamic update which is a basic requirement of many secure network services, such as IDS and firewall.

Keywords: Layer 4 switching; Packet classification; Network security

1. Introduction

With the growing demand of network security, many network security services such as signature-based intrusion detection system (for instance, Snort [18] and NFR [16]) and firewall are widely deployed in recent years. These security services typically involve processing and classifying packets to enforce different protection policies. For example, firewall systems may block any external access to a ftp server, which is for internal use only; to detect possible attacks to a web server, intrusion detection system may search for malicious patterns in the payload of packets with destination port 80. In these security systems, packet classification is first executed to determine the filter in a given classifier database the packet matches, and subsequently the action associated with the matching rule is performed. Several fields of the packet header are generally used for defining a filter. For instance, VPN applications only examine source and destination address fields; firewall queries source address, destination address, protocol, source port, destination port, and flags.

Network security services often delay the transmission of network packets, and consequently degrade the network performance. Due to the increase of network bandwidth, the capability of security systems for processing network packets must be improved accordingly to avoid attacks being undetected. As demonstrated in Ref. [19], although very few packets are undetected when the number of rules is small, the number of undetected packets will increase drastically when the number of detection rules exceeds the maximum processing capability of the system. From this point of view, it is very important to improve the system capability so as to reduce undetected attacks.

Since the time for searching for a matched filter in the classifier database is one of the factors that can dominate the overall performance, it must be reduced as much as possible. Fast packet classification has been investigated by many researchers [2,5,8,9,11,12,15,21,24]. These schemes either suffer from poor search performance, large storage requirement, or are lack of dimension scalability. Other schemes are based on the use of tuple space [22,23,25]. The performance of these schemes is still not satisfactory and their storage requirement is large. It is desirable

[*] Corresponding author. Tel.: +88-692-641-1237; fax: +88-635-724-176.

E-mail addresses: leefy@csie.nctu.edu.tw (F.-Y. Lee), ssp@csie.nctu.edu.tw (S. Shieh), ywlin@csie.nctu.edu.tw (Y.-W. Lin).

[1] http://www.dsns.csie.nctu.edu.tw/ssp.

0140-3664/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2004.05.003

to have a new fast packet classification scheme to increase the throughput of network security services.

In this paper, a packet classification algorithm for improving the performance of network security services is proposed. The new packet classification algorithm utilizes tuple space approach to achieving better search performance than existing multi-dimensional packet classification algorithms, while requiring only little storage expansion. This paper is organized as follows. Section 2 describes the details of the proposed packet classification algorithm and illustrates it with an example. For applications with frequent rule updates, we present a method for minimizing the overhead caused by dynamical update. Section 3 explores the use of proposed scheme in network-based intrusion detection systems. Section 4 gives performance evaluation, complexity analysis and comparison. Finally, the conclusion is presented in Section 5.

2. Kd-tree based tuple space search

In this section, a new packet classification algorithm based on tuple space search is proposed. First, the basic idea of the proposed tuple space search is presented. Then notations and data structures are defined. Our scheme organizes the tuples in the tuple space as a binary search tree, and accordingly the search algorithm for the binary search tree is proposed.

The packet classification problem is to find the matching *filters* in a given *classifier* for a packet P. A filter is an expression that specifies values in the fields of packet header and the action to perform when a packet matches all specifications. A filter f is called a d-dimensional filter if it specifies d fields of the packet header, denoted as $f[1], f[2], \ldots, f[d]$. Each *specification* can specify the prefix, range or exact value in the fields. Note that range specification can be converted to prefix specification using Srinivasan's method [22]. A filter f is said to be a matching filter for a packet P if P can matches all specifications in f. A classifier is a database which contains N filters, ranging between tens and thousands (such as routing table or firewall rule sets).

It has been observed that while a classifier may have many filter rules, the number of distinct prefix lengths in the classifier is usually small. Taking advantage of this feature, the filters in a classifier are partitioned according to the combinations of prefix lengths. If filters with the same combination of prefix lengths are included in the same group, the number of groups would be much smaller than the number of filters. For example, consider the classifier in the traditional router that forwards the packet based on the destination IPv4 address. The classifier can have utmost 32 groups, regardless of the number of filters. For a two-dimensional classifier where each filter specifies the prefix of IPv4 source address and destination address, it can have at most 1024 groups. However, in this case most of the groups may not contain any filter.

Given a d-dimensional classifier, each filter maps to a vector of d integers where the ith integer represents the prefix length of the ith field in the filter. The vector of d integers is called a *tuple* and the set of tuples created by a classifier is called *tuple space*. Since filters mapped to the same tuple have the same number of bits in each field, by concatenating the prefixes of a filter, we can create a hash value using the concatenated bit string for each filter. The hash values are then used to map filters in the same tuple to a hash table. Specifically, consider a filter f mapped to tuple T and its hash value of the concatenation of the prefixes is k. Then, filter f is stored in the kth entry in tuple T's hash table.

To test if a packet P can match any filters in a tuple T, a hash key is created by concatenating the required number of bits from the packet header fields according to T. Then, filter(s) indexed by the hash key is compared with the packet. If the packet matches one of the filters indexed by the hash value, a matching filter is found. The simplest search algorithm based on tuple space search is to probe all the tuples in turn. If each field of a filter is at most W bits, then in the worst case, this approach requires W^d hashes to find a best matching fitler in a d-dimensional classifier.

2.1. Definitions

Tuple T is a vector of d integers which are denoted as $T.vec[1], T.vec[2], \ldots, T.vec[d]$. A filter rule f maps to a tuple T if and only if $\forall i,\ 1 \le i \le d$, prefix length of the ith field of f is exactly $T.vec[i]$. All filter rules map to tuple T are stored in a hash table of T, and $T.ptrHash$ is a pointer to locate T's hash table.

Given a tuple T, the tuple space can be partitioned into three disjointed sets, *LongerTuple* of T, *ShortTuple* of T and *IncomparableTuple* of T. Consider any two tuples T and T_a, $T_a \neq T$ (that is, for some i, $T_a.vec[i] \neq T.vec[i]$). T_a is a tuple in LongerTuple of T if $\forall i,\ 1 \le i \le d$, $T_a.vec[i] \ge T.vec[i]$. If $\forall i,\ 1 \le i \le d$, $T_a.vec[i] \le T.vec[i]$, then T_a is a tuple in ShorterTuple of T. Otherwise, T_a is a tuple in IncomparableTuple of T.

To perform binary search on the tuple space, a binary search tree using the tuples must be constructed. For this purpose, each tuple is associated with a *SuperKey* to determine its location in the binary search tree. SuperKey of tuple T is constructed by concatenating all of the elements in $T.vec$ circularly. A *Discriminator* is used for specifying where to start the circular concatenation. Let $SK_{T.dis}$ denote the SupkerKey of tuple T and discriminator be dis. Then, $SK_{T.dis} = T.vec[dis]T.vec[dis+1], \ldots, T.vec[d]T.vec[1]T.vec[2], \ldots, T.vec[dis-1]$. For example, $SK_{(3,0).1} = 30$ and $SK_{(3,0).2} = 03$ Since there are d elements in $T.vec$, tuple T can have d possible SuperKeys, each starting from $1, 2, \ldots, d$, respectively.

Since SuperKeys are considered integers, they can be sorted. As a result, tuples can be sorted using their

SuperKeys. Given a discriminator *dis*, T_a is *smaller* than T if $SK_{T_a.dis} < SK_{T.dis}$. Otherwise, T_a is *larger* then T. For example, Let discriminator be 1, then tuple (3,0) is larger than tuple (2,2) because $SK_{(3,0).1} > SK_{(2,2).1}$ (30 > 22). If discriminators of all tuples equal to 2, then tuple (3,0) is smaller than tuple (2,2) because $SK_{(3,0).2} < SK_{(2,2).2}$ (03 < 22).

2.2. Construction of Kd-tree

As mentioned in previous sections, filters mapped to the same tuple are stored in a hash table. Tuples can be viewed as multidimensional points in tuple space. Many data structures to organize multidimensional objects are published for last three decades, such as Kd-tree [4], KDB-tree [17], R-tree [13], R + -tree [20], and R*-tree [3]. Among these data structures, Kd-tree is a simple method and more suitable for our need. More information can be found in recent work [1,6,7,10].

In our scheme, each node of Kd-tree stores a tuple T, and two pointers directed to the left and right subtrees. With the definition in Section 2.1, tuples in the left substree of T are the tuples smaller than T; tuples in the right subtree of T are all larger than T. Note that tuples in the left subtree of T belong to either ShorterTuple or IncomparableTuple of T. Tuples in the right subtree of T belong to either LongerTuple or IncomparableTuple of T.

To simplify the illustration of constructing a Kd-tree, assume that both the source address and destination address of a packet are three bits and a simple two-dimensional classifier with only eight filters is given in Table 1. The first field of each filter specifies the prefix of the destination address and the second field specifies the prefix of the source address. Note that '*' means wildcard. For example, '10*' represents all bit strings starting with the first bit '1' and the second bit '0'.

To select the root node for a Kd-tree or a subtree in the Kd-tree, SuperKeys are created to sort tuples. Tuples are sorted according to their SuperKeys and then the middle tuple in the sorted list is selectedas the root node of Kd-tree or a subtree. Tuples smaller then the selected tuple are assigned to the descent left subtree and the rest are assigned to the descent right subtree. Subsequently, to select the root nodes of the descent left and right subtrees, tuples in the descent left and right subtree are sorted again using a new discriminator. In short, $((L \bmod d) + 1)$ is used as the discriminator where L means the level of the tree node to be selected and d represents the dimensions of the classifier. That is, $1, 2, \ldots, d$ are circularly selected as the discriminator. A Kd-tree constructed using the classifier in Table 1 is depicted in Fig. 1. In this example, the tuples are sorted by using 1 as discriminator. The sorted list of tuple is (0,2), (1,2), (1,3), (2,1), (2,2), (3,2), (3,3). The middle tuple (2,1) is selected as the root node of the Kd-tree. Tuples (0,2), (1,2) and (1,3) are designated to the left subtree and tuples (2,2), (3,2) and (3,3) are in the right subtree. Next, let discriminator be 2, tuples in the left and right subtree are sorted again. The sorted sequence will be (0,2), (1,2), (1,3) and (2,2), (3,2), (3,3). Similarly, the middle tuples (1,2) and (3,2) are selected as the root node of the left and right subtree, respectively. By repeating this process, a Kd-tree can be constructed from top to bottom.

Table 1
An example classifier

Rule ID	Filter rule	Mapped tuple
1	(1*,00*)	(1,2)
2	(01*,1*)	(2,1)
3	(10*,01*)	(2,2)
4	(01*,10*)	(2,2)
5	(010,00*)	(3,2)
6	(101,111)	(3,3)
7	(0*,101)	(1,3)
8	(*,10*)	(0,2)

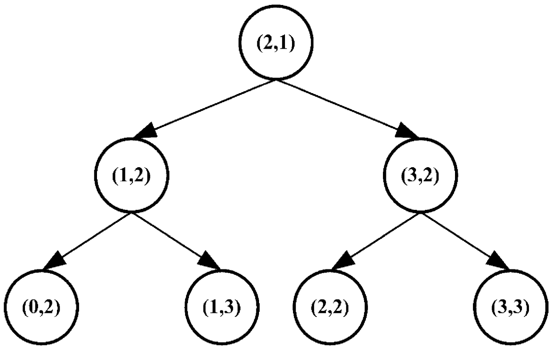

Fig. 1. Kd-tree for Table 1.

Our search algorithm is to perform binary search using Kd-tree. That is, the search algorithm determines the best matching filter by traversing the Kd-tree from the root node to a leaf node. In Section 2.3, a binary search example using the Kd-tree in Fig. 1 is described and followed by another example to show that the simple Kd-tree construction is still not applicable for binary search.

2.3. Search example and mis-judged problem

In this section, a search example is illustrated. A variable *bstMatch* is used to record the best matching filter found by the search algorithm. Consider a packet P with destination address 010 and source address 100. The pair of destination and source addresses is represented as (010,100). Given the classifier illustrated in Table 1 and the Kd-tree in Fig. 1, the searching steps for determine the best matching filter is as follows. The search algorithm first probes the root node of constructed Kd-tree. In this example, tuple (2,1) is the root node and thus it is probed first. To probe tuple (2,1), two prefix bits of the destination address and one prefix bit of the source address are concatenated to construct a hash key. If there is any filter indexed by the same hash value, (010,100) is compared to the filter. In this example, the address pair

(010,100) has the same hash value with filter (01*,1*). Since (010,110) matches the filter (01*,1*), the matching rule (01*,1*) is stored at bstMatch. Afterwards, since the probe in tuple (2,1) returns success, the search algorithm probes the right child, e.g. tuple (3,2). Similar to the way of probing tuple (2,1), the search algorithm uses three bits of the destination address and two prefix bits of the source address to compute the hash value. In this example, the search algorithm cannot find any matching filter. As a result, the search algorithm probes the left child node of (3,2) in the Kd-tree. The search algorithm terminates when it reaches a leaf node in the Kd-tree and reports the rule stored in bstMatch as the best matching filter. In this example, the best matching filter found is (01*,1*).

However, performing binary search using current Kd-tree construction is problematic in some circumstances. As an example, assume that a new rule (010,111) is added and stored in the hash table of tuple (3,3). Consider the problem of searching a matching rule for packet (010,111). The packet will match (01*,1*) at (2,1) but fail at both (3,2) and (2,2). As a result, (01*,1*) will be the best matching rule; nevertheless, (010,011) is a better matching filter than (01*,1*). To solve the problem, markers and pre-computation are included in the proposed scheme. In Section 2.4, details of the solution are described.

2.4. Proposed packet classification algorithm

The proposed packet classification algorithm essentially traverses the Kd-tree to find a best matching filter. However, as illustrated in Section 2.3, the simple construction of Kd-tree caused the search algorithm to report the false best matching filter. In this section, the solution to the mis-judged problem is presented and then the proposed packet classification algorithm is described.

When the search of a tuple fails, the search algorithm searches the left subtree and eliminates all tuples in the right subtree, which may contain the best matching filter. To avoid this mis-judged problem, each tuple should contain information of the rules in its right subtree. For example, tuple (3,2) should have a maker (010,11*) which is generated by the rule (010,111) in tuple (3,3). In this way, to find a best matching filter for packet (010,111), the probe in tuple (3,2) would return success since the search algorithm can find a matching marker (010,11*). Finally the search algorithm can find the best matching filter (010,111) in tuple (3,3).

Let T be a tree node in Kd-tree. Rules of tuple belonging to LongerTuple of T in the right subtree of T will store a marker m in T, where $m[i]$ is the first $T.vec[i]$ bits of the rule. In addition, each rule in the tuples belonging to Incomparable Tuple of T in the right subtree of T will also store markers at T. After that, a best matching filter for each maker m is computed from the filters mapped to tuples in ShortTuple of T. The best matching filter found is stored with the maker m.

As proved in Lemma 1, the right subtree of a tree node T in the Kd-tree can be eliminated when no matching filter or marker is found in T. In contrast, the left subtree cannot be eliminated directly from the search space. Consider the case that a matching marker is found in Tuple T. If the marker is generated by a filter in IncomparableTuple of T, no better matching filter can be found in tuples belonging to IncomparableTuple of T in the left subtree. Otherwise, it violates the conflict-free assumption. If the best matching filter is in the tuple belonging to ShortTuple of T, it has been pre-computed and stored with the marker. Therefore, if the matching marker is generated by a filter in a tuple belonging IncomparableTuple of T, the left subtree can be eliminated from the search space. However, if the marker is generated by a filter in a tuple belonging to LongerTuple of T, the left subtree cannot be eliminated from the search space. This is because a matching filter can still exist in the left substree.

To address this issue, new filters are inserted into the tuple space as follows. For each marker m in a tuple T and each filter f in a tuple belonging to IncomparableTuple of T in the left subtree, if m is conflict with f, then a new filter rule is created to resolve the conflict. The new rule is called a *resolver*. The resolver is generated by taking the longer prefixes in each field from m and f. The resolver is then inserted into the tuple space. It is clear that the resolver will map to a tuple in the right substree of T. Similar to the markers, the best matching filter is computed and stored for each resolver. In this way, as proved in Lemma 2, the left subtree can be eliminated from the search space without considering whether there is a best matching filter in it or not. Fig. 2 shows the pseudo-code for generating the marker. Fig. 3 shows the pseudo-code for pre-compuation and generating resolvers. The search algorithm on Kd-tree is shown in Fig. 4.

Lemma 1. *If packet P does not match any filter or marker at tuple T,P will not match any filter of the tuple in the right subtree of T. Consequently, all tuples in the right subtree of T can be eliminated from search space.*

Proof. We prove this lemma by apagoge. If P matches filter rule f' mapping to a tuple T' in the right subtree of T, then P certainly can match the marker generated by f' in T, which violates the assumption. □

Lemma 2. *If packet P matches a filter rule or marker in tuple T, all tuples in the left subtree of T can be eliminated from the search space.*

Proof. First, we discuss the case that P matches a filter rule f in a tuple T. Since all the filters mapped to a tuple in ShortTuple of T have prefix lengthes shorter than f in all fields, none of them can be a better matching filter than f and, therefore, all these filters can be eliminated from the search space. If P can also match a filter rule f' belonging to a tuple in IncomparableTuple of T in the left subtree, then there is

Procedure 1: Generating Markers

```
1. Construct the tuple space by mapping all filter rules into tuples.
2. Construct a balanced Kd-tree using tuples in the tuple space.
3. for each filter f in the classifier
     Let T denote the tuple that f maps to.
     for each tuple T' in the path from T to the root node of Kd-tree
         if T is in LongerTuple of T'
             Insert marker m in tuple T' for filter f
         end if
         if T is in IncomparableTuple of T'
             Insert markers in tuple T' for f
         end if
     end for
   end for
```

Fig. 2. Construction of the tuple space and markers.

a conflict between f and f'. Due to the conflict-free property, there is a resolver filter for f and f'. Consequently the resolver filter will be the better matching filter than both filters (the resolver has the information of the best matching filter, f or f'). In this case, f' can be eliminated from the search space. Hence, if P matches a filter in a tuple T, the left subtree form the search space can be eliminated from the search space.

Second, we discsuss the case that P matches a marker m generated by a filter rule f. Using precomputation, all

Procedure 2: Pre-computation and Generating Resolvers

```
1. Use Procedure 1 to create markers for each filter f in the given classifier
2. Traverse the Kd-tree from the root node using the sequence of Breadth-First-Search.
3. for each tuple T in the BFS sequence
     for each marker m in tuple T
       Precompute a best matching filter for m from the filters mapped to tuples in ShortTuple of T.
       for each filter f mapped to a tuple belongint to IncomparableTuple of T in the left subtree of T
         if there is conflict between f and m
             Insert a resolver r into the tuple space by taking the longer prefixes in each fields from f
             and m.
             Let $T_1$ denote the tuple that the new created resolver maps to.
             for each tuple $T_2$ in the path from $T_1$ to the root node
                 Insert marker into tuple $T_2$ for the resolver r
             end for
         end if
       end for
     end for
   end for
```

Fig. 3. Precomputation and construction of resolvers.

```
Procedure 3: Search Algorithm
node = root node of the Kd-tree
bstMatch = NULL
repeat
    Let T be the tuple stored in node
    if the probe in tuple T finds a matching marker, resolver or filter
        node = node's right child in the Kd-tree.
        bstMatch = the filter, or the best matching filter stored with the marker or resolver
    else
        node = node's left-child in the Kd-tree.
    end if
until node is a leaf in the Kd-tree.
Return bstMatch
```

Fig. 4. Search procedure.

the tuples in ShortTuple of T can be eliminated from the search space.Next, consider all filters mapped to tuples belonging to Incomparable tuple of T in the left substree. By definition, if P can match f' and m, a resolver is created for m and f'. The resolver is a better matching filter than f' and m, and f' can be eliminated from the search space.
As a whole, if a matching filter or marker is found in a tuple T, the left subtree can be eliminated from the search space. □

With Lemmas 1 and 2, the proposed packet classification algorithm can determine a best matching filter in $O(d \times \log(W))$ hashes, as proved in Theorem 1.

Theorem 1. *Given a packet P and the classifier database, the proposed search algorithm can determine the best matching filter using only* $O(d \log W)$ hashes.

Proof. According to Lemma 1 and Lemma 2, the packet classification algorithm can find the best matching filter by traversing the Kd-tree from the root node to a leaf node. Since the height of the Kd-tree is at most $d \log W$, the number of hashes required is $O(\text{height}) = O(d \log W)$. □

2.5. Dynamic update

So far, the Kd-tree is constructed using only the non-empty tuples. This is true for applications that seldom modify their rule sets. For other applications that frequently change their rule sets, the overhead for dynamically updating is crucial and should be minimized.

The insertion (deletion) of a filter rule may require insertion (deletion) of a new tree node in the Kd-tree, and results in an unbalanced Kd-tree. This degrades search performance because the height of the Kd-tree is no longer $O(d \log W)$. One simple solution is to restructure the Kd-tree when a tree node is inserted or deleted. However, the time for restructuring Kd-tree still constitutes significant overhead. To avoid this, all W^d tuples are used to construct a balanced Kd-tree, i.e. all tuples are included in the Kd-tree regardless whether the tuple is empty or not. Since all tuples are always included in the Kd-tree in the beginning, there is little overhead for inserting or deleting a tree node.

In this way, the first step for inserting or deleting a rule is to locate the tuple that the rule maps to. Then, the rule can be added or deleted from the hash table of the tuple. Afterwards, markers and resolvers are subsequently created or removed correspondingly. This approach significantly reduces the overhead for modifying the rule set.

3. Evaluation and comparison

To evaluate a packet classification algorithm, several criteria including search and update performance, storage cost, scalability of dimensions and filter rules, and flexibility of specifications are examined. In the following context, we assume that a classifier contains N filter rules; every rule has d fields/dimensions; each field has at most W bits.

3.1. Complexity evaluation

Search performance is counted on the basis of number of hashes. Because one tuple is searched by only one hash,

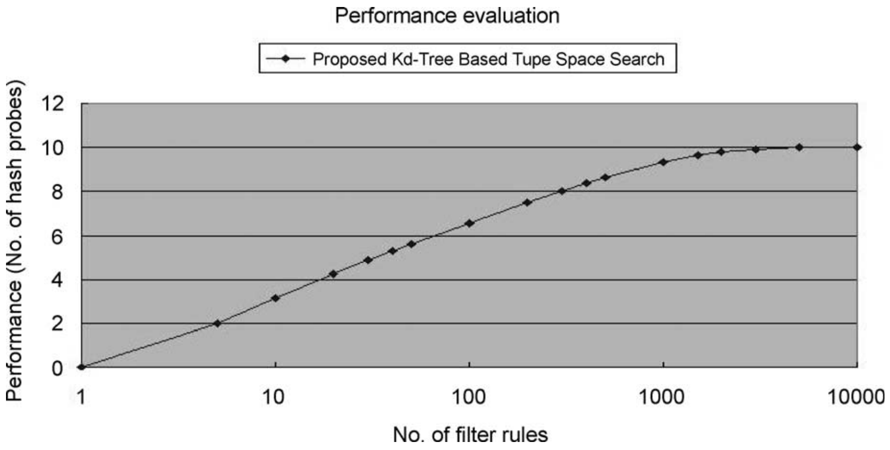

Fig. 5. Performance evaluation of proposed Kd-tree based method.

the time is related to height of Kd-tree. Moreover, the total number of tuples of classifier is W^d tuples. The height of the multidimensional binary tree of tuples is $\log W^d$ or $d \log W$. As a result, complexity of our method is equal to the height of a balanced Kd-tree, which is $O(d \log W)$.

Considering storage complexity of proposed method, it should include cost of filter rules, markers. According to Section 2, filter rules leave markers. In the worst case, a filter rule mapped into the right-most leaf tuple has to leave markers at all tuples in the path from root to its location, resulting in total $d \log W - 1$ markers. Therefore, one filter rule will cost $O(d \log W)$ and totally $O(Nd \log W)$ storage for N filters.

3.2. Experiment on scalability

As the complexity analysis shown in Section 3.1, the proposed scheme can achieve almost constant search time while retaining feasible memory requirement. The advantage of the proposed scheme is particularly important when the number of filters is large. Unfortunately, there is no public available filter sets. To measure the scalability in terms of number of filters, several two-dimensional classifiers are constructed. Each filter in the classifiers specifies the source address and destination address and the address prefixes are randomly selected from the routing table snapshot of Mae-West NAP at March 15, 2002 [14]. In the experiment, the Kd-tree is constructed using only non-empty tuples. Performance with regard to the variation of filter rule number is showed in Fig. 5, where x-axis represents the number of filter rules, and y-axis gives the performance, in terms of number of hashes.

The curve shows that the number of hashes required to find the best matching filter grows slowly with respect to the increasing size of the classifier and ultimately reaches an upper bound. This is because for large classifiers, many filter rules are mapped to the same tuple and the cost for probing a tuple is constant. When all of the tuples are included in the Kd-tree, total $O(d \log W)$ hashes are required. In this experiment, as shown in Fig. 5, the upper bound is 10. $(d = 2, W = 32,$ therefore, $(d \log W) = 2 \times 5 = 10)$.

Table 2
Comparison of packet classification algorithms

Evaluation algorithms	Lookup time	Memory usage	Dimension scalability
Linear search on filter rules	$O(N)$	$O(N)$	Unlimited
Grid-of-tries [21]	$O(W)$	$O(NW)$	2
Cross-producting [21]	$O(dW)$	$O(N^d)$	Unlimited
Bit-parallelism [15]	$O(W \log N)$	$O(NW)$	2
Area-based QuadTree [8]	$O(W)$	$O(NW)$	2
Fat-Inverted segment tree [9]	$O((L+1)W)$	$O(LN^{(1+1/L)})$	2
Segment tree with fractional cascading [24]	$O(\log N)$	$O(N{*}\log N)$	2
Recursive flow classification [11]	$O(d)$	$O(N^d)$	Unlimited
HiCuts [12]	$O(d)$	$O(N^d)$	Unlimited
Linear search on tuple [22]	$O(W^d)$	$O(N)$	Unlimited
Rectangle search [22]	$O(W)$	$O(NW)$	2
Binary search [25]	$O(\log^2 W)$	$O(N \log^2 W)$	2
Extended Grid-of-Trie [2]	$O(W)$	$O(NW)$	Unlimited
Proposed method	$O(d \log W)$	$O(Nd \log W)$	Unlimited

Table 3
IDS policy database (classifier)

Rule ID	Source address	Destination address	Protocol	Source port	Destination port	Flag	Payload content	Log message	Action
R1	*	10.1.1.0/24	tcp	*	79	*	*		log
R2	*	10.1.1.0/24	tcp	*	80	*	/cgi-bin/phf	PHF probe	alert
R3	host A	10.1.1.0/24	tcp	*	6000:6010	*	*	X traffic	alert
R4	*	192.168.1.0/24	tcp	*	143	*	|E8C0FFFF FF|/bin/sh	IMAP buffer overflow	alert
R5	*	10.1.1.0/24	tcp	*	80	PA	/cgi-bin/perl.exe?	CGI-per.exe probe	alert

3.3. Comparison

Table 2 shows the comparison of the proposed method with other packet classification algorithms. Algorithms are listed in the first column and three evaluation criteria, lookup time, memory usage, and dimension scalability are shown in the first row.

First, the lookup time is examined. Most of the existing schemes have either $O(W)$ or $O(\log N)$ time complexity. Four algorithms provide better search efficiency. Time complexity of RFC [11] and HiCuts [12] are constant. The binary search scheme [25] has $O(\log^2 W)$ time complexity. RFC and HiCuts have better time complexity but at the high cost of $O(N^d)$ huge memory space requirement. Excluding RFC and HiCuts, our scheme has the best time complexity $O(d \log W)$ with controlled storage space requirement $O(Nd \log W)$.

Second, we compare the proposed scheme with the two schemes, Rectangle search and Binary search, which are all based on tuple space search. The two schemes can only applied to two-dimensional classifiers, that is $d = 2$. The Rectangle search requires $O(W)$ hashes and the binary search scheme proposed by Warkhede needs $O(\log^2 W)$ hashes. The proposed scheme only requires $O(2 \log W)$ hashes. From the storage perspective, the Rectangle search requires $O(NW)$ and the Binary search requires $O(N^* \log^2 W)$ memory space. In contrast, our scheme only needs $O(N^* 2^* \log W)$ memory space.

4. Application

In this section, our scheme is applied to Network-based IDS (NIDS) to demonstrate its usage. NIDS contains policies and actions to protect computer or network from attacks. As shown in Table 3, the policy database is quite similar to the classifier of a router or a firewall. However, in addition to source and destination addresses of IP header, source and destination ports, other fields, such as protocol, flag, content of packet payload, may be also used in the policies. As a result, the dimension of the policy database of a NIDS is much larger than other applications.

IDS Classifier, as depicted in Fig. 6, is an IDS policy database containing N filter rules, each has d fields. Each filter can be separated into two parts. The first part contains fields in the packet header and the second part specifies the pattern in the packet payload. *Header classifier* is constructed using the first part of each filter rules in the IDS classifier. Let N_h denote the number of distinct filter rules in the header classifier. Each filter rule, called header rule, has d_h fields.

Although every field of packet header can be included in a filter rule, only some of them commonly appear in most of the rules, e.g. destination and source addresses. To reduce the dimension of a filter rule, only the most common fields of the IDS filters are used to construct the header classifier.

Thus, our method should be modified as follows. Unlike original filter rule, header rules derived from IDS classifier are associated with another two pointers

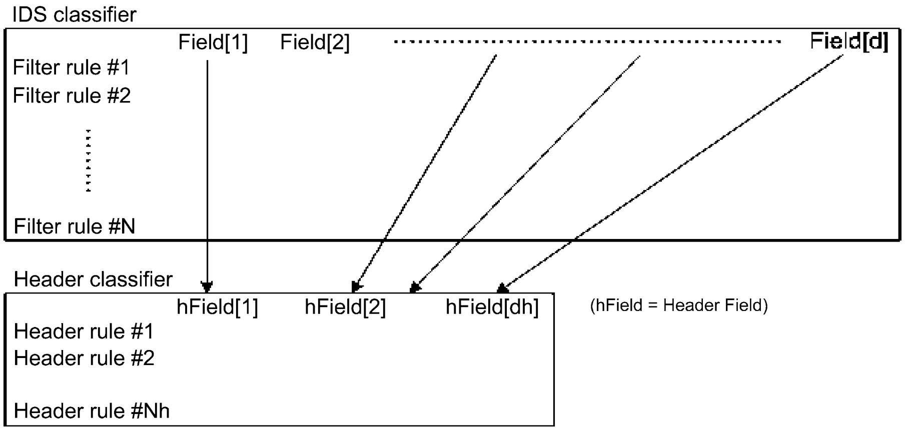

Fig. 6. Transformation of IDS classifier into Header classifier.

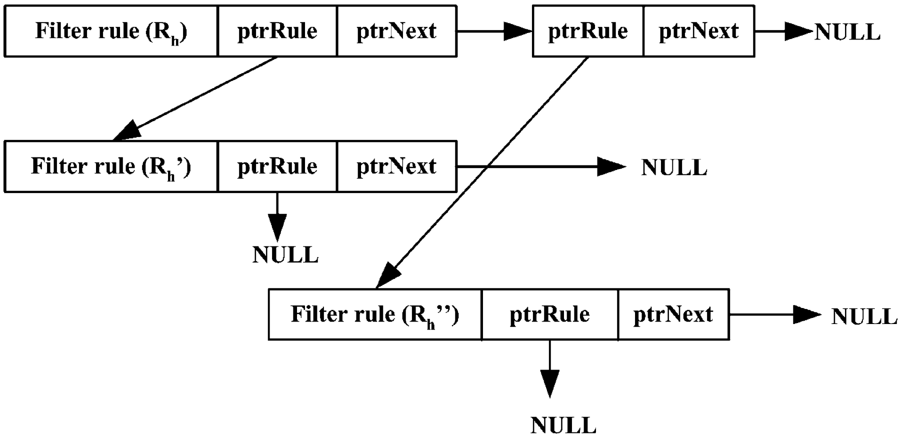

Fig. 7. Data structure for IDS adaptation.

R_h.*ptrRule* and R_h.*ptrNext*. The former pointer directs to other header rule R'_h if R_h is a subset of R'_h. If R_h is a subset of more than one header rule, for every additional rule, two more pointers will be allocated and directed by R_h.*ptrNext*, and new *ptrRule* will point to the additional header rule, as shown in Fig. 7.

As a result, given a packet P, the search algorithm first finds the best matching header filter and compares content of the packet payload with the patterns in the IDS filters. Then, search algorithm traverses all the filter rules using the pointers associated with the best matching header filter. In this way, all the filters in the IDS classifier can be searched very efficiently.

5. Conclusions

In this paper, a solution to packet classification problem using Kd-tree for tuple space search is proposed. Filter rules are grouped into tuples according to their prefix length and tuples are organized as Kd-tree. Each time the search algorithm probes into a tuple, either the left subtree or the right subtree is eliminated. As a result, the proposed search algorithm requires only $O(d \log W)$ hashes to find a best matching filter for a packet, and the storage requirement is only $O(Nd \log W)$.

The way to minimize the overhead for dynamically updating the rules is discussed. Using all tuples to construct the Kd-tree completely reduces the overhead for restructuring the unbalanced Kd-tree. The fast packet classification method can be used to accelerate many security services such as IDS. Since IDS classifier uses many fields to classify packet and only a few fields are commonly used in the filter rules, we suggest that only the commonly used fields are used to construct the header classifier. If a match is found in the header classifier, patterns related to the matching filter are then examined sequentially. In this way, the proposed algorithm provides better performance.

References

[1] H.K. Ahn, N. Mamoulis, H.M. Wong, A Survey on Multidimensional Access Methods, Technical Report of UU-CS-2001-14, 2001.

[2] F. Baboescu, S. Singh, G. Varghese, Packet classification for core routers: is there an alternative to CAMs?, Proceedings of INFOCOM 1 (2003) 53–63.

[3] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles, Proceedings of ACM SIGMOD International Conference on Management of Data, 1990, pp. 322–331.

[4] J.L. Bentley, Multidimentional binary search trees used for associative searching, Communications of the ACM (1975) 509–517.

[5] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry: Algorithms and Applications, Springer, Heidelberg, 1997.

[6] C. Böhm, S. Berchtold, D. Keim, Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases, ACM Computing Surveys (2001) 322–373.

[7] L. Brown, L. Gruenwald, Tree-based indexes for image data, Journal of Visual Communication and Image Representation 9 (4) (1998) 300–313.

[8] M.M. Buddhikot, S. Suri, M. Waldvogel, Space Decomposition Techniques for Fast Layer-4 Switching, Proceedings of Protocols for High Speed Networks, 1999, pp. 25–41.

[9] A. Feldmann, S. Muthukrishnan, Tradeoffs for packet classification, Proceedings of IEEE INFOCOM 3 (2000) 1193–1202.

[10] V. Gaede, O. Gunther, Multidimensional access methods, ACM Computing Surveys 30 (2) (1998) 170–231.

[11] P. Gupta, N. McKeown, Packet classification on multiple fields, Proceedings of ACM SIGCOMM 29 (4) (1999) 147–160.

[12] P. Gupta, N. McKeown, Packet classification using hierarchical intelligent cuttings, Proceedings of Hot Interconnects VII, Stanford, 1999, Also available in IEEE Micro 20(1) (2000) 34–41.

[13] A. Guttman, R-tree: a dynamic index structure for spatial search, Proceedings of ACM SIGMOD (1984) 47–57.

[14] http://www.merit.edu/~ipma/routing_table/

[15] T.V. Lakshman, D. Stiliadis, High-speed policy-based packet forwarding using efficient multi-dimensional range matching, Proceedings of ACM SIGCOMM 28 (4) (1998) 203–214.

[16] Network Flight Recorder, Inc, Network Flight Recorder, 1997.

[17] J.T. Robinson, The K-D-B-tree: a search structure for large multidimensional dynamic indexes, Proceedings of ACM SIGMOD (1981) 10–18.

[18] M. Roesch, Snort—lightweight intrusion detection for networks, Proceedings of the 13th USENIX Systems Administration Conference, 1999, pp. 229–238.

[19] L. Schaelicke, T. Slabach, B. Moore, C. Freeland, Characterizing the performance of network intrusion detection sensors, In Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection, 2003.

[20] T. Sellis, N. Roussopoulos, C. Faloutsos, The R$^+$-tree: a dynamic index for multidimensional objects, Proceedings of Very Large Data Bases, Brighton, England, 1987, pp. 3–11.

[21] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel, Fast and scalable layer four switching, Proceedings of ACM SIGCOMM 28 (4) (1998) 191–202.

[22] V. Srinivasan, S. Suri, G. Varghese, Packet classification using tuple space search, Proceedings of ACM SIGCOMM 29 (4) (1999) 135–146.

[23] V. Srinivasan, A packet classification and filter management system, Proceedings of IEEE INFOCOM (2001) 1364–1473.

[24] C.F. Su, High-speed packet classification using segment tree, Proceedings of IEEE GLOBECOM 1 (2000) 582–586.

[25] P. Warkhede, S. Suri, G. Varghese, Fast packet classification for two-dimensional conflict-free filters, Proceedings of the IEEE INFOCOM (2001) 1434–1443.