Available online at www.sciencedirect.com
ScienceDirect
Computer Networks 51 (2007) 4951–4978
Computer Networks
www.elsevier.com/locate/comnet

Two stage packet classification using most specific filter matching and transport level sharing

M.E. Kounavis *, A. Kumar, R. Yavatkar, H. Vin

Intel Corporation, Intel Research and Development, 1343 NE Alex Way, Apartment 243, Hillsboro, OR 97124, United States

Received 25 January 2006; received in revised form 15 February 2007; accepted 20 August 2007
Available online 31 August 2007

Responsible Editor: D. Stiliadis

Abstract

In this paper we introduce two new concepts to the design of packet classification systems. First, we propose *most specific filter matching* (*MSFM*), an improvement over the well known Cross Producting algorithm [V. Srinivasan, S. Suri, G. Varghese, M. Waldvogel, Fast and scalable layer four switching, in: Proceedings of ACM SIGCOMM, 1998] that significantly reduces the memory requirement of the earlier scheme. Second, we suggest that rules specifying the same source–destination IP prefix pair can be grouped together forming shared sets of transport level fields. This property of *Transport Level Sharing* (*TLS*), which characterizes real world classification databases is exploited for reducing a classifier's memory requirement and for hardware acceleration.

We split the classification process into two stages. First, we perform classification on source–destination IP prefix pairs using the MSFM algorithm. Second, we perform classification on transport level fields exploiting transport level sharing. It is the combination of most specific filter matching and transport level sharing which results in a scheme that requires no more than 11 dependent memory accesses in the critical path independent of the size of the classification database. The memory access bandwidth of our scheme is also bounded when our scheme is accelerated in hardware. Compared to other schemes which involve a small and predictable number of steps in the critical path (e.g., Cross Producting [V. Srinivasan, S. Suri, G. Varghese, M. Waldvogel, Fast and scalable layer four switching, in: Proceedings of ACM SIGCOMM, 1998] or Recursive Flow Classification [P. Gupta, N. McKeown, Packet classification on multiple fields, in: Proceedings of ACM SIGCOMM, 1999]) the combination of most specific filter matching and transport level sharing is associated with the least memory requirement.

Keywords: Switches; Routers; Packet classification; Most specific filter matching; Transport level sharing

1. Introduction

Packet classification is the process of identifying flows from among streams of packets that arrive at routers. Packet classification is still an important and open networking problem because the

* Corresponding author. Tel.: +1 503 712 9222.
E-mail addresses: michael.e.kounavis@intel.com (M.E. Kounavis), alok.kumar@intel.com (A. Kumar).

1389-1286/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2007.08.004

classification step needs to take place as quickly as possible in modern router systems. To ensure that routers can process packets at link speeds ranging from 10 Gbps (e.g., at the network edge) to 10–100 Gbps (e.g., in the network core), the number of memory accesses performed by the classifier must be minimized. Further, the data structures maintained by the algorithm must fit within the small amount of fast memory available in routers. Reducing the memory requirement of a classification scheme is important because fast memory units are usually expensive and because classifiers are expected to support many more rules in the future (e.g., 10,000–100,000 rules) than today. Memory size and access bandwidth is typically limited in the hardware architectures of modern router systems, calling for new techniques for packet classification that minimize the space and time requirement.

In this paper we propose a scheme that can classify packets in a small and predictable number of steps which is independent of the size of a classification database. We introduce two new concepts to the design of packet classification systems. First, we propose most specific filter matching (MSFM) [20–22], a refined version of the well known Cross Producting algorithm [3] that addresses the memory explosion problem associated with the earlier scheme. The basic idea behind MSFM is that significant amount of cross products which are stored as part of a classifier's database can be removed from the database with little penalty to the performance of the classifier. Second, we suggest that rules that specify the same source–destination IP prefix pair can be grouped together forming shared sets of transport level fields. We observe that in real world databases many different sets of source–destination IP prefix pairs are associated with identical sets of transport level fields. This property of Transport Level Sharing (TLS) [20–22], which characterizes real world classification databases is exploited for reducing a classifier's memory requirement and for hardware acceleration.

We split the classification process into two stages. First, we perform classification on source–destination IP prefix pairs using the MSFM algorithm. Second, we perform classification on transport level fields exploiting transport level sharing. It is the combination of most specific filter matching and transport level sharing which results in a scheme that requires no more than 11 dependent memory accesses in the critical path independent of the size of the classification database. While our approach bears similarities with earlier schemes, and especially EGT [12], there are substantial differences in our system design, which result in significant performance benefit. EGT, for instance, finds all possible matching IP prefix pairs for a packet whereas our scheme finds a single match only which is the intersection of the matching IP prefix pairs. When compared to other schemes that perform hierarchical cuttings (e.g., HiCuts [8] and HyperCuts [17]), our scheme has the advantage that it can guarantee that the number of dependent memory accesses in the critical path does not exceed an implementation specific bound. This bound is independent of the classification database used. In contrast, hierarchical cutting schemes build trees of varying heights, where the height of each tree depends on the classification database used. The relative disadvantage of our approach as compared to HyperCuts is that our approach uses more memory. Hierarchical cutting schemes use heuristic algorithms that tradeoff the depth of the tree with the memory requirement and thus can keep the memory requirement of the classifier small. In contrast our approach cannot guarantee that the memory requirement of a classifier does not exceed a space constraint, although it does demonstrate reasonable memory requirement for representative classification databases.

When compared to other schemes which also involve a small and predictable number of steps in the critical path (e.g., Cross Producting [3] or Recursive Flow Classification [7]) our scheme is associated with the least memory requirement. Specifically, the combination of MFSM and TLS demonstrates the least memory requirement for representative classification databases as compared to the state-of-the-art requiring no more than 19–446 KB of memory space for regular size databases of 157–2399 rules. We also observe that the memory requirement of MSFM and TLS scales well with synthetic classification databases of much larger sizes. To reduce the memory requirement of our scheme we exploit properties that characterize real world classification databases used by ISPs and large corporations. We argue that, although such properties characterize a few sample databases, they are likely to hold for most databases.

Unfortunately predictable time performance and reasonable memory requirement do not come for free. Our scheme requires hardware acceleration in order to operate efficiently. The additional hardware required for implementing our scheme is not as costly as a pure TCAM solution though. A hard-

ware unit needed for accelerating the combination of MSFM and TLS can be much smaller and less expensive than a TCAM unit used for implementing the same classifier. For example we found that a hardware unit used for accelerating MSFM and TLS requires 21.6–65.7% of the entries of a TCAM implementing the same classifier. Hardware acceleration helps with reducing the memory access bandwidth requirement of the classifier. Although our scheme can be implemented entirely in software and still require a small and predictable number of dependent memory accesses, the memory access bandwidth requirement is higher without acceleration (16 times higher in our implementation).

The update process is currently time consuming since it requires the creation of a number of different lookup tables, the reordering of rules, the discovery of shared sets of transport level fields and the creation of the data structures used by the hardware accelerator. However, our approach can be modified to support fast incremental updates at the expense of the optimality of the classification data structure and lookup algorithm.

The paper is structured as follows. In Section 2 we formulate the classification problem. In Section 3 we discuss the characteristics of classification databases which we exploit in our design. In Section 4 we describe related work. In Section 5 we describe the design of our algorithms and discuss how this design is derived from first order principles. In Section 6 we evaluate our scheme. Finally, in Section 7 we provide some concluding remarks.

2. Problem formulation

A packet classifier consists of a rule database, a lookup algorithm and an update algorithm. A router that supports classification stores its set of rules $R = \{R_i, i \in [1,n]\}$ into the classification database. Each rule comprises a set of fields $F_i = (F_i^1, F_i^2, \ldots, F_i^k)$, a priority level P_i and an action A_i. Fields specify ranges of values associated with portions of the TCP/IP header (i.e., the source IP address, the destination IP address, or the protocol field). IP address ranges are usually specified as prefixes. Port number ranges are arbitrary. A rule or a subset of the fields of a rule is often called a *filter*. An example of rules found in a router database is given in Table 1.

The are two instances of the classification problem: The 'single match' classification problem can be stated as follows: Given a set of rules R in an ordered list L, what is the action $A(p)$ associated with the highest priority rule that matches a given packet p? Another instance of the classification problem, the 'multiple match' problem can be stated as follows: Given a set of rules R in an ordered list L, what are the actions $A_i(p)$ and identifiers $i(p)$ associated with *all* the rules that match a given packet p? In this document we present a solution to the single match classification problem. Extending our approach for solving the multiple match classification problem is the subject of future work.

Table 1
Example of transport level sharing

Src. IP address	Dest. IP address	Src. port	Dest. port	Action	Priority
147.101.*	*	*	ftp	Permit	1
147.101.*	*	*	10–50	Deny	2
132.*	145.*	*	ftp	Permit	3
132.*	145.*	*	10–50	Deny	4

The single match packet classification problem is more difficult than IP routing [14] because a packet may match with rules at arbitrary priority levels. We believe that the need for classifying packets in a small and predictable number of steps calls for new techniques and methodologies for solving the classification problem. The approach we follow in this paper is to design a scheme based on the properties of classification rules.

3. Characteristics of classification rules

3.1. Characteristics of IP prefix pairs

Each rule in a database contains the specification of a source and a destination IP prefix. Each IP prefix can be specified as a wildcard, a range or an exact value. For example, the IP prefix 132.* in the example of Fig. 1 is a range that includes all IP addresses

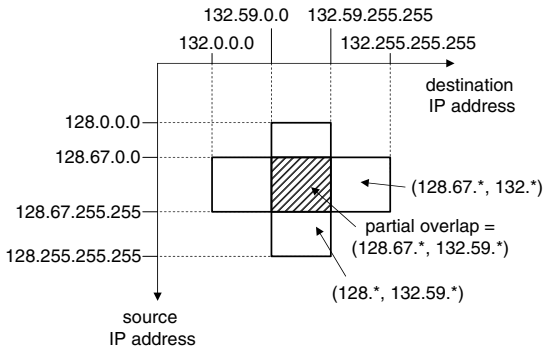

Fig. 1. Partially overlapping filters.

from 132.0.0.0 up to 132.255.255.255. Based on this observation, IP prefix pairs may represent rectangles, lines or points in the two-dimensional IP address space. In addition, such filters may overlap with each other either partially or completely. By 'overlap' of two filters we mean the set of all possible combinations of field values which define points in the space included in each of the two filters. Two filters overlap 'partially' if their overlap is different from any of the two filters. For example, the filters (128.67.*, 132.*) and (128.*, 132.59.*) shown in Fig. 1 overlap partially. Their overlap is equal to the filter (128.59.*, 132.67.*).

Binary prefix pairs can form, in the worst case, $n \cdot (n-1)/2$ partial overlaps, which is $O(n^2)$, where n is the number of binary prefix pairs in a set. In the special case where binary prefixes are IPv4 prefixes the upper bound $n \cdot (n-1)/2$ is only achieved if $n \leqslant 33$. This happens because an IP prefix can only have 33 lengths. If $n > 33$ the number of partial overlaps is still $O(n^2)$ in the worst case, although the upper bound $n \cdot (n-1)/2$ cannot be achieved in practice. An example of a filter structure that creates the highest possible amount of partial overlaps is shown in Fig. 2a. Each filter F_i shown in the figure creates an overlap with every other filter F_j, $i \neq j$. Another structure that creates $n^2/4$ partial overlaps is illustrated in Fig. 2b. The structure of Fig. 2b can be achieved for any realistic value of n. In the structure of Fig. 2b each filter F_i shown in the figure creates an overlap with half of all other filters F_j, $i \neq j$.

It has been observed that in real databases the amount of partial overlaps formed between filters is significantly smaller than the theoretical worst case $n \cdot (n-1)/2$ as well as the $n^2/4$ bound. Databases usually demonstrate filter structure which is substantially different from the structures of Figs. 2a and 2b. The filter structure characterizing real databases is shown in Fig. 2c. While in the structures of Figs. 2a and 2b every filter forms partial overlaps with at least half of all other filters, in the realistic structure of Fig. 2c overlaps are created from only a small subset of the filters.

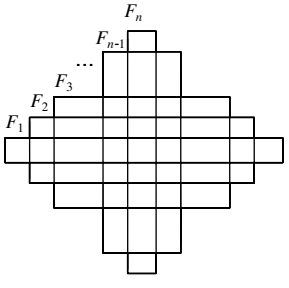

Fig. 2a. Worst-case partial filter overlaps.

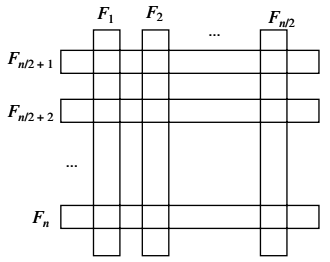

Fig. 2b. $n^2/4$ partial filter overlaps.

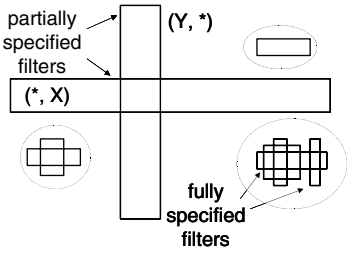

Fig. 2c. Realistic partial filter overlaps.

There are three sources of partial overlaps between IP prefix pairs in classification databases. First, partial overlaps may be created between partially-specified filters. We define the set F_P of the partially-specified filters of a database L as the set

$$F_P(F) = \{F_i = (S_i, D_i) : F_i \in F, S_i = * \vee D_i = *\}, \quad (1)$$

where by $\vee$ we mean the 'OR' logical operator and F is the set of IP prefix pairs of database L. Partially-specified filters have the wildcard in the source or destination IP address dimensions (i.e., they are filters of the form $(*, X)$ or $(Y, *)$). Similarly we define the set F_F of the fully-specified filters of a database L as the set:

$$F_F(F) = \{F_i = (S_i, D_i) : F_i \in F, S_i \neq * \wedge D_i \neq *\}, \quad (2)$$

where $\wedge$ is the 'AND' logical operator.

Each filter having the wildcard in the source IP address dimension creates a unique partial overlap with all filters having the wildcard in the destination

IP address dimension. A second source of partial overlaps includes overlaps created between fully-specified filters (i.e., filters that do not include the wildcard). Fully-specified filters may overlap with each other either fully or partially. Third, overlaps may be created between fully- and partially-specified filters.

It has been observed in earlier studies (in Ref. [16]) that partially-specified filters are the main source of partial overlaps in classification databases. In addition, partially-specified filters represent a small fraction of the total number of filters in regular size and large classification databases ranging from 0% to 20%. This happens because network administrators usually specify rules that apply to the traffic exchanged between specific IP address domains. Fully-specified filters create an insignificant amount of overlaps between each other ranging from 0% to 4%. This happens because in most cases fully-specified filters are segments of straight lines or points (i.e., the source or destination field represents a host or an important server) and because network clients and servers are usually connected to different subnets (i.e., they are associated with disjoint IP prefixes). Because of these reasons the number of partial filter overlaps observed in classification databases is small and has been found to be between 0% and 1.6% of the $n \cdot (n-1)/2$ theoretical worst case and 0–3.2% of the $n^2/4$ bound in representative classification databases.

We believe that, as network administration policies become more complex and applications requiring end-to-end Quality of Service (QoS) become more popular, these observed properties of classification databases will continue to hold in the future. First, it is likely that classification rules that do not specify the wildcard (*) in the source or destination IP address dimension will continue to represent the majority of rules in classification databases. This is likely to happen because such rules have the flexibility of expressing more refined administration policies between specific network domains. Hence such rules can be used for expressing a wider set of access control and QoS policies than the rules which apply to all the traffic sent to or from network domains. Second, we believe that partial filter overlaps created between fully-specified filters will continue to represent a small fraction of the total amount of filter overlaps. There are two reasons for this. First, it is likely that clients and servers will continue to be connected to different subnets. This is a standard network administration practice for security. Second, fully-specified filters may continue to be represented by segments of straight lines or points in the two-dimensional space. Because of all these reasons we believe that our solution can be useful in the future just as it is applicable in representative classification databases which are in use today.

3.2. Characteristics of transport level fields

In the Internet there are thousands of routers but relatively only a few, commonly used applications. As a result, only a small number of port number values and ranges are usually specified in rules. Earlier studies on the properties of classification databases report the fact that source destination IP prefix pairs are associated with a small number of transport level fields [12] and that transport level fields form sets which are being shared between many different source–destination IP prefix pairs [16].

Table 1 shows an example of transport level sharing. The rules at priority levels 1 and 2 in the table are associated with the same IP prefix pair (i.e., 147.101.*, *). The first rule specifies the destination port number to be equal to ftp whereas the second rule specifies the destination port number to be in the range 10–50. The rules at priority levels 3 and 4 specify a different IP prefix pair (i.e., 132.*, 145.*) but the same transport level fields as rules 1 and 2. In this case the set {ftp, 10–50} of transport level fields is shared between the sets of rules 1–2 and 3–4. In the example of Table 1 we also observe that the relative priority and action associated with each entry of the shared set of transport level fields is the same in each occurrence of the set.

We define the set $B(L)$ of the shared sets of transport level fields of a classification database L as the set of all unique instances of sets of transport level fields B_i associated with the same source–destination IP prefix pair, where the relative priority and action of each entry of every set B_i is the same in each occurrence of this set.

Transport level sharing is a property which characterizes typical classification databases because network administrators specify the same transport level fields associated with popular applications or common network management practices in many different IP prefix pairs. The number of entries in unique instances of sets of transport level fields has been investigated in [16] and has been found to be between 11% and 42% of the number of rules in a database.

3.3. Implications

3.3.1. Classification on the source and destination IP address dimensions

Since each packet represents a point in the two-dimensional space, it may be contained within the geometrical space defined by one or more IP prefix pairs in a database. Therefore, a packet may match multiple IP prefix pairs within a database. We specify the set F_M of all matching IP prefix pairs $F_i = (S_i, D_i)$ of a packet p associated with source IP address S and destination IP address D as

$$F_M(p) = \{F_i = (S_i, D_i) : F_i \in F, S_i \prec S \wedge D_i \prec D\}, \tag{3}$$

where by $\prec$ we mean the 'prefix of or equal to' operator, and F is the set of IP prefix pairs of a database. The expression $X \prec Y$ is true when X is a prefix of Y or $X = Y$.

Identifying the highest priority rule requires comparing the transport level fields associated with a packet's matching IP prefix pairs with the appropriate fields contained in the packet header. Clearly, the larger the number of filters that a packet may match with, the greater the latency of identifying the highest priority rule that matches the packet may be. Finding all possible matches for a packet in the source–destination IP address dimensions may require a significant amount of memory accesses. In addition, the number of worst case memory accesses required may vary from one database to another.

An alternative approach involves finding a single match for a packet in the source–destination IP address space. This match should be equal to the smallest intersection of filters that cover the point in the space representing the packet. We call this filter the *most specific* filter for the packet. The approach we follow in this paper is to split the packet classification process into two stages and in the first stage find the most specific filter for a packet in the source and destination IP address dimensions. We formally define the most specific filter F_{MS} of a packet p as

$$F_{MS}(p) = \bigcap_{i=1}^{|F_M(p)|} F_i, F_i \in F_M(p), \tag{4}$$

where by $\cap$ we mean the intersection operator and by $|x|$ we mean the cardinality of set x.

An example of a most specific filter for a packet is shown in Fig. 3. A packet p shown in the figure is

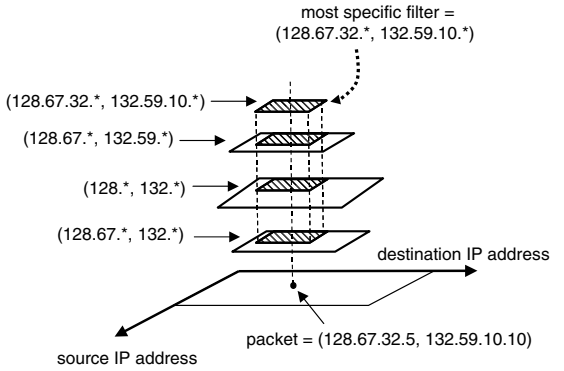

Fig. 3. Most specific filter of a packet.

associated with source IP address equal to 128.67.32.5 and destination IP address equal to 132.59.10.10. The packet matches four filters. One approach to classify the packet would be to search the rules associated with each of the packet's matching IP prefix pairs. An alternative approach would be to search the rules which cover the most specific filter which is equal to (128.67.32.*, 132.59.10.*).

To support most specific filter matching, classifiers need to include all possible intersections between IP prefix pairs into their data structures. We argue that most specific filter matching is a feasible technique because the number of partial IP prefix pair overlaps in databases is smaller than the theoretical worst case. As it has been discussed earlier, the number of partial overlaps which IP prefix pairs form in the two-dimensional space is small. Therefore the memory requirement of algorithms that find the most specific filter of packets may not be prohibitive in practice.

3.3.2. Classification on transport level fields

One of the benefits from transport level field sharing is compression. Each shared set of transport level fields needs to be stored only once for all IP prefix pairs that share this set. As a result the total amount of entries required for storing transport level fields can be very small.

A more significant benefit from transport level sharing is that classification based on transport level fields can be easily implemented in hardware. Since the total number of entries in unique sets of transport level fields is small these sets can be stored in some specialized hardware unit capable of checking whether a packet matches the entries of these sets in parallel. Such hardware unit would be similar to a Ternary CAM (TCAM) unit [10] with the exception

that it would only store the shared sets of transport level fields of databases instead of the fields of all rules in databases. The traditional TCAM design can be extended in many different ways to support transport level range matching operations. One way to modify the traditional TCAM design is to introduce binary comparator circuits in each entry of the hardware unit in order to support efficient multidimensional range matching checks. Another approach is to use an off-the-shelf TCAM unit after expanding port number ranges to prefixes. A range of alternative hardware designs is presented in Section 5.

The cost of a specialized hardware unit would be significantly higher if the transport level fields of rules were stored without taking into account sharing. The cost of a specialized unit used for storing the unique instances of sets of transport level fields may be viable however, even for large classification databases. For example, we found that a hardware unit used for storing the unique sets of the transport level fields of the databases we experimented with requires 21.6–65.7% of the entries of a hardware unit that stores the transport level fields of all rules in the same databases. For our experiments we used the same databases as in Ref. [16]. The range 21.6–65.7% which we report here differs from the range 11–42% reported in [16] because port number ranges in our case are expanded to prefixes.

Fig. 4 illustrates our framework for two stage packet classification using most specific filter matching and transport level sharing. The classification process is split into two stages. In the first stage classification is performed in software on the source and destination IP address dimensions. The first stage determines the most specific filter that matches with a packet. This most specific filter is associated with at least one shared set of transport level fields. Stage 1 returns an ordered list of pointers to shared sets of transport level fields, which are used for classification in the remaining dimensions. In stage 2, classification is performed in hardware. A selected subset of the transport level field sets stored in the hardware unit is activated in a single step. The second stage returns the action associated with the highest priority entry among the transport level field sets identified in Stage 1.

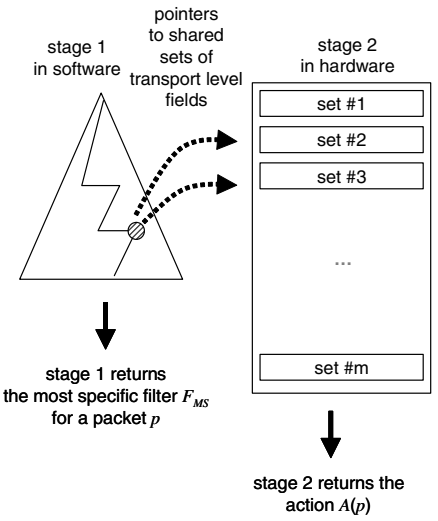

Fig. 4. Two stage packet classification using most specific filter matching and transport level sharing.

4. Related work

Existing packet classification algorithms [1] can be grouped into four classes: trie-based algorithms, hash-based algorithms, parallel search algorithms, and heuristic algorithms. Throughout this discussion, we use n to denote the number of rules in a classification database, k to denote the number of fields (i.e., dimensions), and w to denote the maximum length of the fields (in bits).

Trie-based algorithms [2,3,5,6] build hierarchical radix tree structures where once a match is found in one dimension a search is performed in a separate tree linked into the node representing the match. Examples of such algorithms are the Grid-of-Tries [3] and Area-based Quad Tree (AQT) [5] algorithms. Trie-based algorithms require, in worst case, as many memory accesses as the number of bits in the fields used for classification. Multi-bit trie data structures are more efficient from the perspective of the number of memory accesses required. However, these data structures incur significantly higher memory space overhead. In general, trie-based schemes work well for single-dimensional searches. However, the memory requirement of these schemes increases significantly with increase in the number of search dimensions.

Hash-based algorithms [9,11] group rules according to the lengths of the prefixes specified in different fields. The groups formed in this manner are called ‘tuples’. Hash-based algorithms perform a series of hash lookups one for each tuple to identify the highest priority matching rule. Tuple space search has $O(n)$ storage and time complexity. Hash-based algorithms, in the worst case, require as many memory

accesses as the number of hash tables, and the number of hash tables can be as large as the number of rules in a database. As a result, hash-based techniques do not scale well with the number of rules. An optimized hashing technique, referred to as rectangle search [9], reduces the lookup time complexity from O(n) to O(w) in two dimensions. However, to support lookups in more than two dimensions, the algorithm may still require a significant number of memory accesses.

Parallel algorithms formulate the classification problem as an n-dimensional matching problem and search each dimension separately. In some algorithms [4], when a match is found in a dimension, a bit vector is returned identifying the matches. The logical AND of the bit vectors returned from all dimensions identifies the matching rules. Such bit vector techniques are associated with O(n) memory accesses in the lookup process. Fetching a single bit vector or an aggregate bit vector (as described in [13]) can be memory access intensive, especially in cases where the classification database contains more than a few thousand rules. Another parallel search technique called Cross Producting Table [3] reduces the lookup time complexity to O(kw) where k is the number of fields and w is maximum length of the fields. However, this technique increases the worst case storage complexity to O(n^k) making it impractical. An improvement over the Cross Producting technique called distributed Cross Producting of field labels is described in [15]. This technique reduces the memory requirement of the classifier at the expense of the lookup time. Distributed Cross Producting can be accelerated, however, using bloom filters.

A fourth category of algorithms includes heuristic algorithms that exploit the structure and redundancy in the rule-set (e.g., Recursive Flow Classification [7] and HiCuts [8]). Most of the heuristic algorithms proposed to-date are associated with very low lookup time complexity O(k); however, they impose significant memory space requirement O(n^k). Hence, these algorithms are suitable for single- or two-dimensional searches, but their space requirement makes them unsuited for the more common five-dimensional searches. A variation of [8] called HyperCuts [17] appears to be associated with both smaller lookup time and space requirement as compared to HiCuts [8]. Like HiCuts [8], however, the worst case lookup time requirement of HyperCuts depends of the database used.

5. Algorithm design

5.1. Cross Producting

The most specific filter of a packet can be found with well known techniques such as Set Pruning Tries [2], Cross Producting Table [3] and Recursive Flow Classification [7]. Cross Producting and Recursive Flow Classification are the fastest techniques because they can search the source and destination IP addresses of packets in parallel as opposed to sequentially. Among the two, Cross Producting can impose the least memory requirement because this technique employs a single stage of parallel searches and hence requires a single lookup table. The algorithm which we propose in this paper is built upon the Cross Producting technique due to its speed and potential for memory requirement reduction.

Cross Producting is shown in Fig. 5. In each of the source and destination IP address dimensions a Longest Prefix Matching (LPM) search takes place. Each search returns an index. The two indexes $\langle I_1 \rangle$ and $\langle I_2 \rangle$ returned from the stage of parallel LPM searches are combined into a third index $\langle I_1 I_2 \rangle$ which is used for accessing a table of cross products. By the term 'cross product' we mean a filter F_i with source IP address equal to the source IP address of some filter F_j of a database and destination IP address equal to the destination IP address of some other filter F_k of the database, not necessarily equal to F_j. A formal definition of the set of cross products F_C of a database is given below

$$\begin{aligned} F_C(F) = \{F_i = (S_i, D_i): & \\ & \text{there exist filters} \\ & F_j = (S_j, D_j) \in F, F_k = (S_k, D_k) \in F \\ & \text{for which } S_i = S_j \wedge D_i = D_k\}, \end{aligned} \tag{5}$$

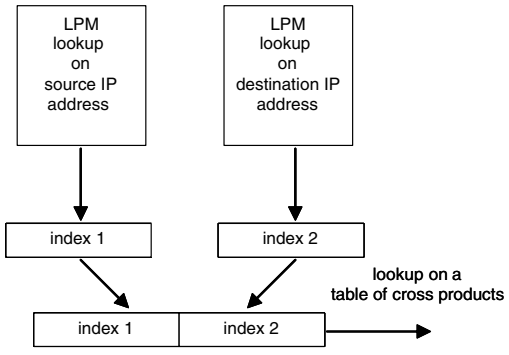

Fig. 5. Cross Producting.

where F is the set of IP prefix pairs of the database. The table of cross products can be implemented in many different ways such as a hash table.

The reason why the Cross Producting table technique places all possible cross products into a lookup table is because the stage of LPM searches may return a filter that does not exist in the classification database. To explain why this may happen we show an example in Fig. 6. In the example of Fig. 6, three filters exist in a classification database: (128.67.*, 132.59.*), (128.67.32.*, 121.45.5.*) and (125.12.12.*, 132.59.10.*). The most specific filter of the packet (128.67.32.5, 132.59.10.10) is the filter (128.67.*, 132.59.*). The stage of LPM searches returns the prefix 128.67.32.* from the source IP address dimension and the prefix 132.59.10.* from the destination IP address dimension. This happens because these prefixes are the longest matching prefixes in the classification database for the given packet. The filter which results by combining the returned source and destination IP prefixes (128.67.32.*, 132.59.10.*) is clearly a filter that is not included in the classification database. Such filter is a cross product. By calculating all possible cross products of a database and placing them into a lookup table, the Cross Producting technique guarantees that a match is always found for every packet by the stage of parallel LPM searches.

5.2. Improving Cross Producting

The main disadvantage of the Cross Producting technique is the increase in the classification data structure size caused by the need to store all cross products of IP prefix pairs. Cross products can be in the worst case as many as the square of the number of rules in a classification database. In representative databases we experimented with, the number of cross products is significantly higher than the number of rules. For example, in the access control list 'ACL3' shown in Tables 3 and 4 below there are 431 unique source IP prefixes and 516 unique destination IP prefixes resulting in 222,396 cross products. Because of this reason the memory requirement of the Cross Producting technique for ACL3 is almost 1MB.

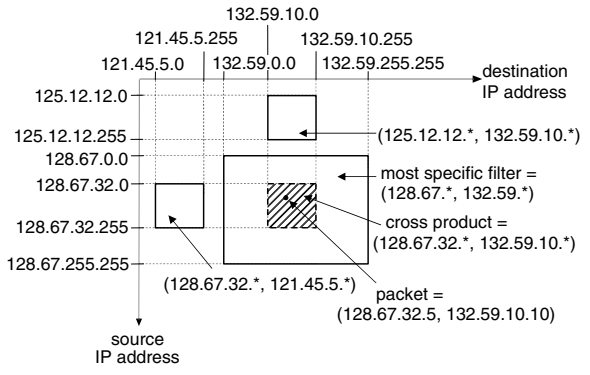

Fig. 6. Cross Producting returning a non-existent filter.

In this section we argue that the size of the lookup table used by the Cross Producting technique can be significantly reduced by observing that from among the many cross products only a few really need to be placed in the lookup table. A significant amount of cross products can be removed the lookup table with little penalty to the performance of the classifier.

A first group of cross products which can be removed from the lookup table are those for which there is no filter in the database apart from (*,*) that contains them. We call these cross products 'not covered' since they are only contained into (*,*). We formally define the set F_{NC} of the not covered cross products of a database as

$$F_{NC}(F) = \{F_i \in F_C(F), F_i \notin F \cup F_I(F): \text{there is no filter } F_j \in F \text{ for which } F_j \neq (*,*), F_j \prec F_i\}, \quad (6)$$

where by $\cup$ we mean the union operator, F is the set of IP prefix pairs of a database and $F_I(F)$ is the set of intersections of the IP prefix pairs of F. The operator $\prec$ for filters means 'completely contains or is equal to'. For two filters X, Y for which $X = (x_1, x_2)$ and $Y = (y_1, y_2)$, the expression $X \prec Y$ means $x_1 \prec y_1$ and $x_2 \prec y_2$. The reader should notice that in the definition of the set $F_{NC}(F)$ we do not include cross products which are database filters or filter intersections. The reason why we exclude such filters from definition (6) is to help with proving the correctness of our algorithm.

A cross product which belongs to the set $F_{NC}(F)$ for some set of filters F is shown in the example of Fig. 7. In the example of Fig. 7 the filters (128.67.32.*, 121.45.5.*) and (125.12.12.*, 132.59.10.*) form the cross product (125.12.12.*, 121.45.5.*) which is only contained into the two-dimensional space (*,*). This cross product is a not covered cross product. Filters (128.67.32.*, 121.45.5.*) and (125.12.12.*, 132.59.10.*) also form the cross product (128.67.32.*, 132.59.10.*) which

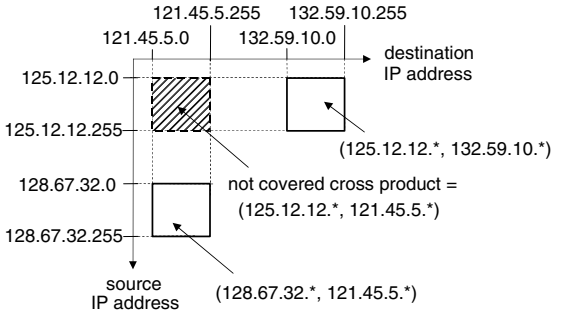

Fig. 7. Example of a not covered cross product.

is not covered but omitted from the figure for the sake of simplicity.

The reason why not covered cross products can be removed from the lookup table is because they are only contained into $(*,*)$. If the stage of LPM searches does not return an entry in the lookup table, then this means that the cross product which results from the LPM searches is not covered. Hence the most specific filter for a given packet is $(*,*)$. So, one could design a refined version of the Cross Producting technique where not covered cross products are removed from the lookup table and the algorithm returns $(*,*)$ if an entry in the table is not found.

There is also another group of cross products which can be removed from the lookup table. These are the cross products which are only covered by partially-specified filters or filter intersections that are partially-specified. We call these cross products 'partially covered'. The term should not be confused with 'partially overlapping'. Partially covered cross products are called that way because they are only contained into filters which specify one of the two dimensions. We formally define the set F_{PC} of the partially covered cross products of a database as

$$F_{PC}(F) = \{F_i \in F_C(F), F_i \notin F \cup F_I(F) \cup F_{NC}(F): \text{there is no filter } F_j \in F_F(F \cup F_I(F)) \text{ for which } F_j \prec F_i\}, \quad (7)$$

where F is the set of IP prefix pairs of a database, $F_I(F)$ is the set of intersections of the IP prefix pairs of F, and $F_{NC}(F)$ is the set of not covered cross products of F. The reader should also notice that in the definition of the set $F_{PC}(F)$ given above we exclude cross products which are database filters, filter intersections or not covered cross products. Such definition helps with proving the correctness of our algorithm.

A cross product which belongs to the set $F_{PC}(F)$ for some set of filters F is shown in the example of Fig. 8. In the example of Fig. 8 the filters $(128.67.32.*, 121.45.5.*)$ and $(125.12.12.*, 132.59.10.*)$ form the cross product $(125.12.12.*, 121.45.5.*)$ which is contained into the partially-specified filter $(125.12.12.*, *)$. This cross product is a partially covered cross product. Thefilters $(128.67.32.*, 121.45.5.*)$ and $(125.12.12.*, 132.59.10.*)$ also form the cross product $(128.67.32.*, 132.59.10.*)$ which is omitted for the sake of simplicity.

The reason why partially covered cross products can be removed from the lookup table is because they are only contained into partially-specified filters. If the stage of LPM searches does not return an entry in the lookup table, then this means that the cross product which results from the stage of parallel LPM searches is either partially covered or not covered. Hence, the most specific filter for a given packet is either a partially-specified filter or $(*,*)$. If the most specific filter for a packet is partially-specified but not $(*,*)$, this filter can be potentially identified from the stage of parallel LPM searches as explained below. On the other hand, if the most specific filter for a packet is $(*,*)$, this filter can be returned once all other searches fail.

So, one could refine the Cross Producting technique even further by removing both the not covered and partially covered cross products from the lookup table. If an entry is not found in the lookup table, then the algorithm checks whether the most specific filter for a given packet can be determined by the result of the LPM searches. If the most specific filter cannot be determined, the algorithm returns $(*,*)$.

The only cross products which cannot be removed from the lookup table without significantly

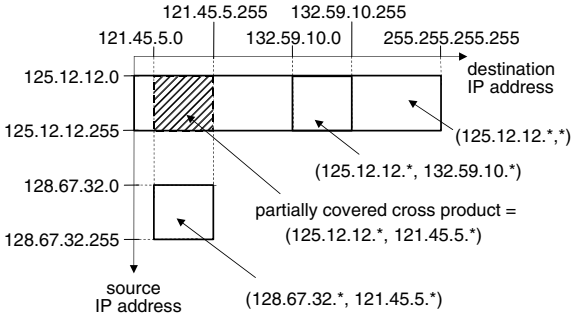

Fig. 8. Partially covered cross product.

penalizing the performance of the classifier are those which are neither not covered nor partially covered. We call these cross products 'fully covered'. We define the set F_{FC} of the fully covered cross products of a database as

$$F_{FC}(F) = F_C(F) - F_{NC}(F) - F_{PC}(F). \quad (8)$$

From definitions (6)–(8) one can show that the set of fully covered cross products $F_{FC}(F)$ of a set of filters F is the set which includes the following elements:

- all fully-specified filters in F,
- all filter intersections that are fully-specified,
- all filters which are formed by combining the source and destination IP prefixes of different IP prefix pairs and which are contained into fully-specified filters in F or filter intersections that are fully-specified.

Theorem 1. *Let F be a set of IP prefix pairs. Then, the set of fully covered cross products $F_{FC}(F)$ associated with F is given by*:

$$F_{FC}(F) = F_F(F) \cup F_F(F_I(F)) \cup F_{IN}(F)1, \quad (9)$$

where the set $F_{IN}(F)$ introduced in Eq. (9) is given by the expression:

$$F_{IN}(F) = \{F_i \in F_C(F), F_i \notin F \cup F_I(F): \text{there exists a filter } F_j \in F_F(F \cup F_I(F)) \text{ for which } F_j \prec F_i\}. \quad (10)$$

Theorem 1 tells us that the set of fully covered cross products $F_{FC}(F)$ associated with a set of filters F can be found as the union of three terms. The first term $F_F(F)$ represents the fully-specified filters in F. The second term $F_F(F_I(F))$ represents the intersections of filters in F which are fully-specified. The third term, which we denote as $F_{IN}(F)$, represents filters which are neither elements of the set of fully-specified filters $F_F(F)$ nor elements of the set of intersections that are fully-specified $F_F(F_I(F))$. The elements of this set $F_{IN}(F)$ are contained into at least one fully-specified filter or fully-specified filter intersection. The elements of the set $F_{IN}(F)$ are formed by combining the source and destination IP prefixes of different IP prefix pairs. We call these filters 'indicator filters' because any of these filters, if returned after the stage of parallel LPM searches, can point to the most specific filter for the packet which is classified. Because of the fact that indicator filters do not belong to any of the sets $F_F(F)$ or $F_F(F_I(F))$ (i.e., they are not filters of a database but only artificially formed regions) none of the indicator filters can be the most specific filter of a packet. The role of indicator filters in our scheme is further explained in Section 5.3. The Proof of Theorem 1 is given in the Appendix.

5.3. Most specific filter matching (MSFM)

5.3.1. Algorithm description

The observations discussed above motivate the design of a new two-dimensional scheme for finding the most specific filter for a packet. We call our scheme 'most specific filter matching' (MSFM). Like Cross Producting, our scheme uses parallel LPM searches in the source and destination IP address dimensions. Unlike Cross Producting, our scheme does not include all cross products into a lookup table. Instead, our scheme includes entries associated with fully covered cross products into a primary lookup table and entries associated with partially-specified filters into two secondary lookup tables. A separate entry is stored for the entire two-dimensional space $(*, *)$.

The first of the two secondary tables contains entries associated with all partially-specified filters that have the wildcard in the destination IP address dimension (i.e., filters of the form $(X, *)$). The second secondary table contains entries associated with all filters having the wildcard in the source IP address dimension (i.e., filters of the form $(*, Y)$). The primary table is implemented as a hash table.

The MSFM algorithm builds two trie data structures for the source and destination IP prefixes of filters in order to perform parallel LPM searches on these prefixes. Each prefix is marked as associated with a partially- or fully-specified filter or both. The lookup process of the MSFM algorithm, shown in Fig. 9, is as follows:

Step 1: Two parallel LPM searches are performed in the source and destination IP address dimensions. For each packet each search returns the longest matching prefix associated with a partially-specified filter and the longest matching prefix associated with a fully-specified filter. Each prefix is mapped into a separate index. Both searches return a total of four indexes.

Step 2: After the step of parallel LPM searches is complete lookup tables are accessed in

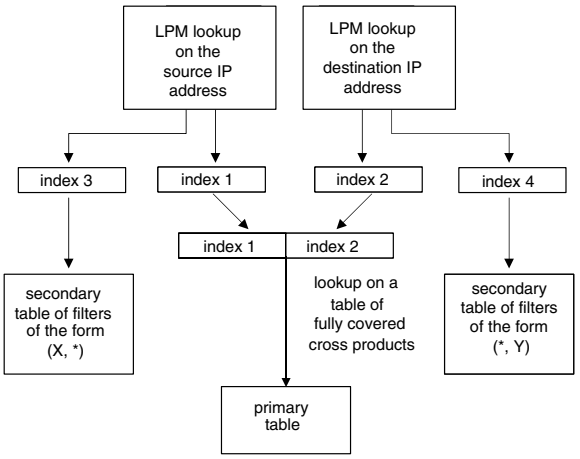

Fig. 9. Lookup Process in the MSFM algorithm.

parallel. Let $\langle I_1 \rangle$ and $\langle I_2 \rangle$ be the indexes associated with fully-specified filters returned from the source and destination LPM searches of Step 1. The combined index $\langle I_1 I_2 \rangle$ is used for accessing the primary lookup table. Similarly, let $\langle I_3 \rangle$ be the index associated with a partially-specified filter returned from the LPM search on the source IP address dimension. The index $\langle I_3 \rangle$ is used for accessing the first of the two secondary tables. Finally, an index $\langle I_4 \rangle$ associated with a partially-specified filter returned from the LPM search on the destination IP address dimension is used for accessing the second secondary table.

Step 3: In this step the results from the parallel queries on the lookup tables are examined. If the query on the primary table returns a match, this match indicates the most specific filter of the packet. If the query on the primary table does not return a match then there may be a case that either the first or the second secondary table returns a match. In this case, the match returned is the most specific filter of the packet, which is a partially-specified filter. If none of the lookup tables returns a match, then the most specific filter for the packet is the two-dimensional space $(*,*)$. In this case the algorithm returns $(*,*)$.

5.3.2. Correctness

The correctness of MSFM scheme can be formally proven using the following theorem:

Theorem 2. *Let $F_{MS} = (S_{MS}, D_{MS})$ be the most specific filter for a packet p. Then the following statements are true*:

(*i*) *if $S_{MS} \neq * \wedge D_{MS} \neq *$, then a stage of parallel LPM searches on the source and destination addresses of p returns a filter $F_{LPM} = (S_{LPM}, D_{LPM})$ which is either equal to F_{MS} or it is completely contained into F_{MS}. The relationship between filters F_{MS} and F_{LPM} can be formally expressed as $F_{MS} \prec F_{LPM}$.*

(*ii*) *if $S_{MS} \neq * \wedge D_{MS} = *$, then the prefix returned from an LPM search on a tree that contains the source IP prefixes of partially-specified filters (i.e., filters of the form $(X,*)$) is exactly equal to S_{MS}.*

(*iii*) *if $S_{MS} = * \wedge D_{MS} \neq *$, then the prefix returned from an LPM search on a tree that contains the destination IP prefixes of partially-specified filters (i.e., filters of the form $(*,Y)$) is exactly equal to D_{MS}.*

The Proof of Theorem 2 is given in the Appendix. Theorem 2 tells us is that if the most specific filter for a packet is fully-specified then the stage of parallel LPM searches returns either this most specific filter or a filter that is completely contained into the most specific filter. In the second case, by definition (9) and statement (i) of Theorem 2, the filter returned is an indicator filter. The entry associated with this indicator filter can be set to point to the most specific filter for the packet which is classified. The MSFM algorithm is correct in this case because all possible indicator filters for given packets are identified and placed into the primary table.

If the most specific filter for a packet is partially-specified then the stage of parallel LPM searches returns exactly this most specific filter according to statements (ii) and (iii) of Theorem 2. To make sure that the stage of parallel LPM searches returns the longest matching prefixes of partially-specified filters, MSFM marks each prefix as being associated with a fully- or partially-specified filter or both. If the query on the primary lookup table does not return a match then the most specific filter for a packet must have the wildcard in the source or destination IP address. In this case either the first or the second secondary table may return a match. If no table returns a match, then this means that the only filter which covers the packet is the two-dimensional space $(*,*)$.

5.3.3. Example

We further illustrate how MSFM works by an example. Let us consider the filters A, B, C, D and E shown in Table 2 and Fig. 10. The combinations of the source and destination IP prefixes of these filters (including the wildcard $*$) form 20 different regions in the two-dimensional space. Among these regions, filters A–E and are illustrated with thick continuous lines in Fig. 10. The other nine regions R_1–R_9 shown in the figure correspond to cross products. Regions R_1–R_9 are illustrated with thin dotted lines in Fig. 10. For the sake of simplicity the figure omits cross products that have the wildcard ($*$) in the source or destination IP address dimensions.

The smallest filter that completely covers the regions R_1, R_3 and R_4 in the example of Fig. 10 is filter A, which is the entire two-dimensional space $(*,*)$. The points in the space that are contained into the regions R_1, R_3 and R_4 (but not into regions D and E) are also contained into the two-dimensional space $(*,*)$. Hence R_1, R_3 and R_4 are not covered cross products. If the entries for the regions R_1, R_3 and R_4 are removed from the lookup table, the scheme still works provided that there exists a separate entry for the filter $(*,*)$ which is checked in the case the stage of parallel LPM searches fails to return any valid result.

Table 2
The filters A–E in the example of Fig. 10

Filer	Src. IP address	Dest. IP address
A	*	*
B	147.101.10.*	*
C	128.67.*	132.59.*
D	125.12.12.*	132.59.10.*
E	128.67.32.*	121.45.5.*

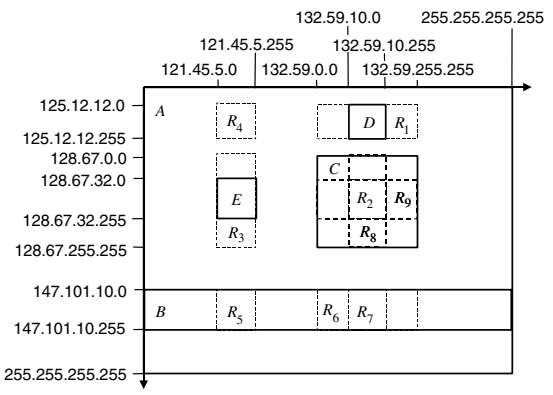

Fig. 10. An example of source–destination IP prefix pairs.

Similarly, we observe that the regions R_5R_6 and R_7 can be removed from the lookup table as well. R_5R_6 and R_7 are completely covered by filter B, which is a partially-specified filter having the wildcard in the destination IP address. In addition, filter B is the filter with the smallest coverage area that completely covers regions R_5, R_6 and R_7. Hence R_5, R_6 and R_7 are partially covered cross products. We observe that an LPM search in the source IP address dimension returns the source IP prefix of filter B for all packets representing points in the regions R_5, R_6 and R_7. Therefore, the results of a stage of parallel LPM searches can still identify filter B even if the cross products R_5, R_6 and R_7 are not included in the lookup table. As a result, the cross products R_5, R_6 and R_7 do not need to be placed in the lookup table.

The only regions which cannot be removed from the lookup table in the example of Fig. 10 are regions R_2, R_8 and R_9. These regions are completely covered by a fully-specified filter (C). Region R_2 for example, is formed by the destination IP prefix of filter D and the source IP prefix of filter E. Therefore, for any packet that represents a point included into R_2 there needs to be an entry in the lookup table associated with this region. Region R_2 is a fully covered cross product and, more specifically, an indicator filter.

5.3.4. Time and space requirements

The lookup time complexity of the MSFM algorithm is $O(w)$ where w is the number of bits in the source or destination IP address fields. To find indicator filters MSFM needs to scan all source and destination IP prefixes in the lookup table. Finding indicator filters is associated with $O(n^4)$ complexity in our current implementation, where n is the number of filters in the database. This operation, however, is not part of our fast lookup algorithm and hence it does not penalize the performance of the classifier. Indicator filters are determined only during the times the classification data structure is built. The storage complexity of the MSFM algorithm is $O(n^2)$, where n is the number of IP prefix pairs in a database. The storage complexity of MSFM is the same as the complexity of Cross Producting. In reality, however MSFM requires significantly fewer KB of memory to store its lookup data structure.

For example, for the set of filters shown in Fig. 10, the Cross Producting technique needs to

place entries for 20 regions of the two-dimensional space in the lookup table. MSFM requires to place entries for 8 regions only (i.e., filters A–E, R_2, R_8 and R_9) resulting in 60% reduction of the lookup data structure size. In real large databases the reduction is even greater for two reasons. First partially-specified filters represent a small percentage of the total number of filters in large databases as discussed in Section 3. As a result, the size of the secondary tables is expected to be small. Second, most fully-specified filters are segments of straight lines or points. As a result, the number of indicator filters that need to be placed in the primary table is expected to be small. The primary table contains at least as many entries as the number of partial overlaps formed between filters. This number, however, is expected to be lower than the theoretical worst case as discussed earlier.

Table 3 shows the sizes of the primary and secondary tables for the databases we experimented with. The first three databases (i.e., ACL 1–ACL 3) are access control lists from large ISPs whereas the fourth database (i.e. ACL 4) is an access control list from a corporate intranet. The number of databases we present in this paper is small due to the fact it is hard to get permission from ISPs and large corporations to disclose access control list content. Despite the fact that the number of databases we present in this paper is small, we believe that our results are valid for three reasons: First, ACL 1–ACL 4 come from real edge routers or corporate intranets. ACL 3 in particular comes from the network of a very well known large ISP. Second, the properties of ACL 1–ACL 4 which justify the efficiency of our approach have been observed in several other ACLs which cannot be disclosed in this paper. Third, we have been able to associate these properties with standard network administration practices and therefore argue that these properties are likely to characterize many different databases.

Table 3
Memory requirements of the Cross Producting and MSFM schemes (in KB)

	Number of rules	Cross Producting (total)	MSFM (primary table)	MSFM (tries and secondary tables)	Data structure size reduction (%)
ACL 1	754	87.7	74.4	11.2	2.4
ACL 2	607	160.1	79.8	14.4	41.2
ACL 3	2399	900.2	167.4	35.1	77.5
ACL 4	157	9.5	0.3	3.7	57.9

From the results of Table 5 it is evident that MSFM results in significant reduction of the lookup data structure size, which is as high as 77.5% in the third database. Since there were no collisions in the entries of the hash tables we constructed we used exactly one processor word to represent each table entry. Each processor word is four bytes in our implementation. In the general case, if some collisions occur for some databases, then the prefix index values which result in such collisions can be changed so that collisions are avoided. Avoiding collisions is possible in our scheme for two reasons: First, the filters of each database are known in advance or at least during the rule update phase. Second, the prefix indexes which are used for accessing the hash tables of MSFM are set to arbitrary values.

Fig. 11 shows the distribution of the number of indicator filters for the four databases we experimented with. Most fully-specified filters or fully-specified filter intersections do not contain indicator filters. The fraction of filters that do not contain indicator filters ranges from 57.6% in ACL 3 to 92.6% in ACL 4. This is the main reason why MSFM results in significant reduction of the lookup data structure size as shown in Table 3. It is also evident from Fig. 11 that there exist a small number of filters which include a significant number of indicator filters. For example, only two filters include as many as 18018 indicator filters in ACL 1. Hence, it is evident that the MSFM scheme may not work efficiently if some filters in databases include excessively many indicator filters. In the databases we experimented with, such filters are only but a few. These filters are characterized by small prefix lengths either in the source or the destination dimension spanning more than one IP address

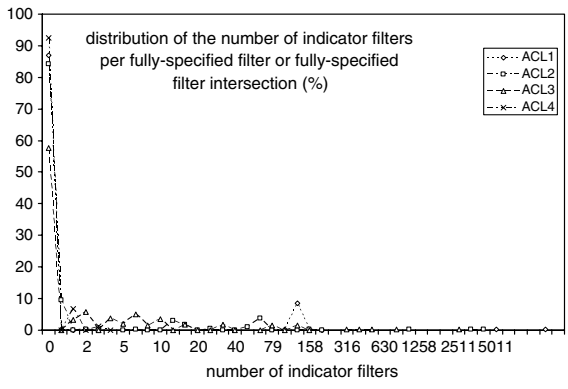

Fig. 11. Distribution of the number of indicator filters.

domains. These filters span multiple IP address domains because they describe generic administration policies. We call these filters 'wide filters' in this document. We found that there are 2 wide filters in ACL1, 8 wide filters in ACL 2, 15 wide filters in ACL 3 and no such filters in ACL 4.

Our observations on the distribution of the number of indicator filters, shown in Fig. 11 motivate an improvement on the most specific filter matching scheme. We can specify a bound on the number of indicator filters which can be stored for a fully-specified filter or a fully-specified filter intersection. Filters which include more indicator filters than a specified bound are removed from the primary table and placed in a separate lookup data structure. This lookup data structure can be implemented as a Cross Producting table. The lookup process can be modified allowing each LPM search to return three indexes as opposed to two with little penalty to the performance of the classifier. The first two indexes returned are associated with a fully- and a partially-specified filter as in the regular MSFM algorithm description. The third index, if returned, is associated with a filter removed from the primary table because of containing excessively many indicator filters. Such filter is a wide filter. The Cross Producting table containing wide filters is accessed in parallel with the primary and secondary tables. The lookup process determines the most specific filter for the packet as in the regular MSFM algorithm description. The order in which tables are accessed is the following. First, the primary table is accessed. If the primary table contains no entry for the packet which is classified, then the wide filter table is accessed. If the wide filter table contains no entry for the packet which is classified, then the secondary tables are accessed. If the queries on the secondary tables fail as well, then the most specific filter for the packet is (*, *).

Table 4 indicates that the improved MSFM scheme results in significant decrease of the memory requirement of access control lists ACL 1 and ACL 2. For ACL 1, the data structure size reduction becomes 69.3%. For ACL 2, the data structure size reduction becomes 62.5%. Finally, for ACL 3, the data structure size reduction becomes 80.0%. The bound we used for distinguishing between regular and wide filters was 200 indicator filters.

Apart from large databases like ACL 1–ACL 4 there exist some for which the memory requirement of MSFM is close to Cross Producting. Specifically, we observed that for some small databases (less than 100 rules) the impact of MSFM on the memory requirement of the classifier is diminished. These databases are used in corporate intranets for local access control and contain exceedingly many partially specified or wide filters. For these databases the overall memory requirement is small. Hence a two stage scheme like the one proposed in this paper is feasible for these databases as well.

Table 4
Memory requirements of the Cross Producting and the improved MSFM schemes (in KB)

		Cross Producting (total)	Improved MSFM (primary table)	Improved MSFM (tries and remaining lookup tables)	Data structure size reduction (%)
ACL 1	754	87.7	15.7	11.2	69.3
ACL 2	607	160.1	45.6	14.5	62.5
ACL 3	2399	900.2	144.4	35.3	80.0
ACL 4	157	9.5	0.3	3.7	57.9

5.4. Transport level sharing (TLS)

5.4.1. Design issues

To design an efficient scheme for checking transport level fields we exploit the fact that most rules that specify the same source destination IP prefix pair occupy adjacent priority levels in real databases (i.e., these rules are inserted sequentially in the databases). We find that, not only there is sharing characterizing the sets of transport level fields of the rules specifying the same source–destination IP prefix pair, but also there is sharing characterizing the sets of the rules specifying the same source–destination IP prefix pair at adjacent priority levels. In what follows, we use the term 'rule-set' to refer to the set of rules that specify the same source–destination IP prefix pair. By 'distance' of a rule-set we mean the maximum difference between the priority levels of any two rules in a rule-set divided by the number of rules in the rule-set. Obviously, the rule-set distance is one measure that indicates the proximity of the priority levels of the rules in the same rule-set. Table 5 shows the distance distribution for all rule-sets in the four databases we experimented with.

The results of Table 5 indicate that most rule-sets have distance equal to 1 in real databases. The intuition behind this observation is that network administrators add rules specifying the same source–destination pair (i.e., the same network

Table 5
Rule-set distance distribution

	Rule-sets having distance equal to one (%)	Rule-sets having distance between 1 and 100 (%)	Rule-sets having distance greater than 100 (%)
ACL 1	99.1	0.9	0
ACL 2	97.7	1.6	0.7
ACL 3	86.4	7.2	6.4
ACL 4	80.6	18.4	1

path) sequentially at consecutive priority levels. We also observe in Table 5 that a small fraction of rule-sets have distance greater than one ranging between 1 and 100. Another fraction of the rule-sets have very large distance greater than 100, which is in the order of the number of rules in the database.

Rule-sets that have distance which is small but greater than one are created by interleaving the rules associated with the same application but with different source–destination IP prefix pairs. For example rules may be interleaved as shown in Table 6.

To exploit transport level sharing we reduce the distance for some rule-sets. To reduce the distance for some rule-sets (and to increase the proximity of the priority levels of their rules) we change the order of some elements in the database so as to make sure that interleaved rules occupy adjacent priority levels in databases. The reordering algorithm which is part of the update process is the following: We move each new rule 'up' or 'down' the priority list as long the rules below or above specify a different IP prefix pair and do not overlap. We stop when we find an adjacent rule that has the same source–destination IP prefix pair as the new rule. If we do not find any such rule the new rule is inserted at its original priority level. Reordering reduces the distance for some sets significantly as shown in Table 7.

Table 6
Interleaved rules

Src. IP address	Dest. IP address	Src. port	Dest. port	Action	Priority
128.59.*	132.12.*	*	www	Permit	n
147.102.*	12.45.*	*	www	Permit	$n+1$
134.22.*	221.34.*	*	www	Permit	$n+2$
128.59.*	132.12.*	*	ftp	Permit	$n+3$
147.102.*	12.45.*	*	ftp	Permit	$n+4$
134.22.*	221.34.*	*	ftp	Permit	$n+5$
128.59.*	132.12.*	*	telnet	Permit	$n+6$
147.102.*	12.45.*	*	telnet	Permit	$n+7$
134.22.*	221.34.*	*	telnet	Permit	$n+8$

Table 7
Rule-set distance distribution after reordering

	Rule-sets having distance equal to one (%)	Rule-sets having distance between 1 and 100 (%)	Rule-sets having distance greater than100 (%)
ACL 1	100	0	0
ACL 2	99.6	0	0.4
ACL 3	98.7	0	1.3
ACL 4	99	0	1

In what follows we present an example of how our reordering algorithm works: Consider the rule (128.59.*, 132.12.*, *, ftp, permit) at priority level $n+3$ in Fig. 7. For this rule the reordering algorithm would consider moving its priority level up by one position and placing it at level $n+2$. The rule at level $n+2$ (134.22.*, 221.34.*, *, www, permit) does not overlap with (128.59.*, 132.12.*, *, ftp, permit). Several fields of these two rules (e.g., their source IP prefixes) are disjoint. Because of this reason the algorithm considers moving the priority level of the rule (128.59.*, 132.12.*, *, ftp, permit) further up by one position and placing the rule at level $n+1$. The rule at level $n+1$ (147.102.*, 22.45.*, *, www, permit) does not overlap with (128.59.*, 132.12.*, *, ftp, permit). Because of this reason the algorithm considers moving the priority level of the rule (128.59.*, 132.12.*, *, ftp, permit) up by even one more position and placing the rule at level n. The rule at level n, specifies the same source destination IP prefix pair as (128.59.*, 132.12.*, *, ftp, permit). Because of this reason the algorithm stops. The rule is placed at priority level $n+1$ and the rules at priority levels $n+1$ and $n+2$ move by one position down at levels $n+2$ and $n+3$ respectively.

Since the majority of the rule-sets have distance equal to one and source–destination IP pairs exhibit transport level sharing, it is expected that the sets of transport level fields resulting from grouping together the rules that have the same source–destination prefix pairs and occupy adjacent priority levels in the database are also shared. This is an interesting observation which we exploit in our design. The size distribution of sets of transport levels fields formed from adjacent rules that specify the same IP prefix pair after reordering is shown in Fig. 12. The level of sharing associated with these sets is shown in Table 8.

The most specific filter returned from stage 1 is associated with a union of shared sets of transport level fields. These sets correspond to the rules that

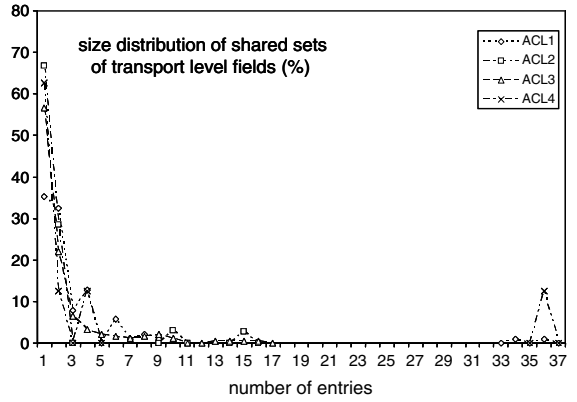

Fig. 12. Size distribution of shared sets of transport level fields.

Table 8
Level of sharing associated with sets of transport level fields

	Number of source–destination IP prefix pairs	Number of unique sets of transport level fields	Number of entries in unique sets of transport level fields
ACL 1	426	102	316
ACL 2	527	35	68
ACL 3	1588	186	437
ACL 4	98	8	47

cover the most specific filter returned by MSFM. While the entries of sets of transport level fields may have arbitrary priorities inside their unions, the entries of sets constructed by grouping rules at adjacent priority levels are associated with consecutive priority levels inside their unions. As a result, it is sufficient to associate the most specific filter returned from stage 1 with an ordered list of shared sets of transport level fields. In one of our implementations (see below) 11 pointers of 16 bits each are sufficient for describing the ordered lists of sets of transport level fields returned from MSFM after reordering. These pointers occupy a total space of 6 words and can be fetched in a single memory access time.

5.4.2. Algorithm description

In this section we describe our classification algorithm for transport level fields. We call our algorithm *Transport Level Sharing (TLS)* from the basic property of transport level fields which our algorithm exploits. TLS consists of two stages: A preprocessing stage and a lookup stage. The preprocessing stage takes place every time the classifier's data structure is rebuilt. The lookup stage takes place every time a packet enters a router.

Preprocessing:

Step 1: Rules that specify the same source–destination IP prefix pair and occupy adjacent priority levels in the database are grouped together.

Step 2: The transport level fields of the groups of rules formed by Step 1 are identified.

Step 3: The unique instances of all sets formed by Step 2 are identified and placed in a specialized hardware acceleration unit capable of checking the entries of each set in parallel.

Step 4: Each of the filters returned from the stage of source–destination IP address classification is associated with an ordered list of pointers to unique instances of sets of transport level fields. These sets correspond to the rules that cover each filter returned by MSFM.

Lookup:

Step 1: MSFM is executed first. MSFM returns an ordered list of pointers that activate a selected subset of the shared sets of transport level fields stored in the hardware unit. The ordered list of pointers is sent to the hardware unit and the hardware unit returns the action associated with the highest priority match.

TLS is associated with O(1) lookup time complexity provided that (i) the list of pointers returned from MSFM can be fetched with a small number of memory accesses; (ii) a specialized hardware unit can check the entries of each selected set in parallel. The first assumption is valid in real world databases because each packet is usually covered by no more than 7 IP prefix pairs [16] and because the distance of most rule-sets is equal to 1. To investigate the validity of the second assumption we evaluated a number of possible alternative ways to accelerate Transport Level Sharing in hardware. Our alternatives include 'single bin' and 'multiple bins' range matching accelerators and off-the-shelf TCAMs.

5.4.3. Hardware acceleration of TLS

TLS can be accelerated in hardware in three different ways. In one implementation a specialized hardware acceleration unit consists of sets of comparator entries called 'bins'. Each entry in a bin consists of a number of binary comparators that can

perform range matching checks in the source and destination port number dimensions and exact value matching checks in the protocol field dimension. A priority encoder attached to a bin returns an index and action value associated with the highest priority match. In this implementation each bin stores a different set of transport level fields. The MSFM algorithm returns an order list of pointers to bins and the hardware unit activates the bins selected by MSFM from its stored set. The hardware unit returns the highest priority match from the order specified by MSFM. Such hardware unit is illustrated in Fig. 13. The main advantage of this approach is that transport level fields are stored in the hardware acceleration unit at the preprocessing phase. As a result MSFM needs to return only pointers to transport level fields as opposed to the values of the transport level fields themselves. Hence the memory access bandwidth requirement of this implementation can be kept small as shown in the evaluation section. The main disadvantage of this approach is the complexity of the priority resolution circuit which needs to be attached to the bins of the hardware unit.

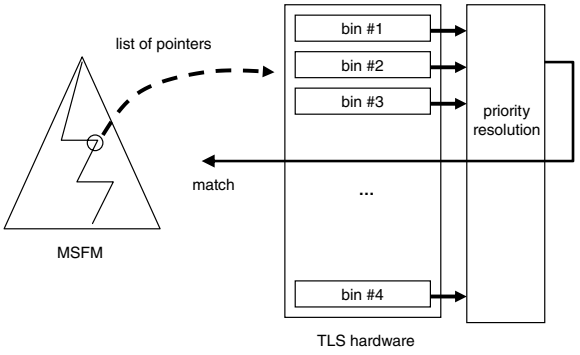

Fig. 13. Multiple bins hardware implementation.

An alternative approach sacrifices the memory access bandwidth efficiency for the ease of implementation. The custom hardware acceleration unit, shown in Fig. 14, helps with checking a limited number of entries in parallel (i.e., the entries of a single bin of the previous approach) but contains a single 'bin' only. The arrows in the figure show how a specific hardware building block (e.g., a range matching unit) is being implemented. The basic hardware building block of this unit is a comparator circuit. The comparator circuit consists of a subtractor that compares the key passed (i.e., a port number value) against an upper or lower bound value. In this implementation transport level field entries are copied from an external memory unit into the accelerator during the lookup process. Although copying transport level field entries can ideally take place in a single step after MSFM, this approach imposes higher memory access bandwidth requirement than the previous approach because it requires the physical transfer of port number range values and protocol fields from an external memory unit into the custom hardware.

A third implementation choice reduces both the memory access bandwidth and the complexity of implementation at the cost of the space required for storing transport level field sets. This implementation uses an off-the-shelf TCAM unit for storing transport level fields.

This approach is illustrated in Fig. 15. Each unique instance of a shared set of transport level fields is placed in an off-the-shelf TCAM unit. The port number ranges of the transport level field entries in each set are expanded into prefixes. Before

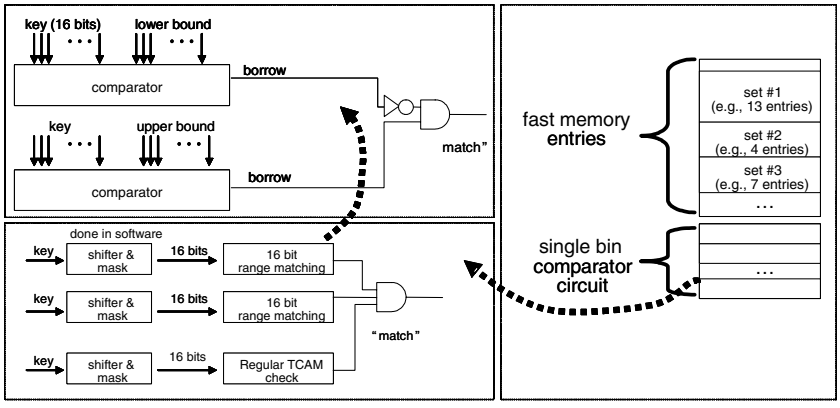

Fig. 14. Single bin hardware implementation.

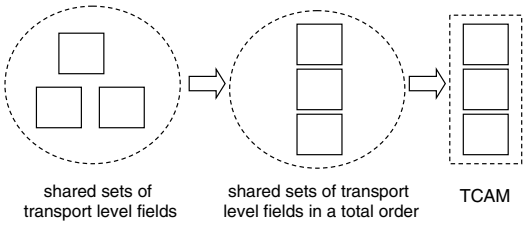

Fig. 15. Off-the-shelf TCAM implementation.

inserting transport level field sets into the TCAM a total order is defined for the sets of transport level fields satisfying the way entries are prioritized in each MSFM match. In this way the TCAM unit can determine the highest priority match for a given packet.

5.4.4. Defining a total order for sets of transport level fields

To create a total order between shared sets of transport level fields one can use an easy to implement greedy algorithm, which operates as follows: The algorithm initializes a totally ordered set of transport level field sets O to one list of sets of transport level fields returned from MSFM: $O \leftarrow O^1 = \{L_1^1, L_2^1, L_3^1, \ldots, L_m^1\}$ chosen arbitrarily. By L_j^i we denote the jth set of transport level fields of the i^{th} ordered list returned from MSFM. The algorithm sets a variable called *insert-index* equal to 1. For every possible ordered list of sets that can be returned from MSFM O^i, the totally ordered set $O = \{L_1, L_2, L_3, \ldots, L_n\}$ is updated as follows: Let k be the smallest index such that

$$insert\_index \leqslant k \leqslant n \tag{11}$$

and

$$L_k = L_1^i. \tag{12}$$

If such k exists then the algorithm sets *insert-index* to be equal to $k+1$. Otherwise the algorithm inserts the set of transport level fields L_1^i at position *insert-index* and increments the index *insert-index* by one. The previous step is repeated until all elements of the ordered list O^i have been taken into account. When the algorithm finishes processing the ordered list O^i, it sets the variable *insert-index* equal to 1 again.

An example is shown in Fig. 16. In the example of Fig. 16, MSFM returns one of four lists of transport level field sets: $\{1,2,6,5\}$, $\{3,7,5\}$, $\{6,5\}$ or $\{6,2\}$. In this example we use integers to refer to transport level field sets for the sake of simplicity. In the first step the algorithm sets the totally

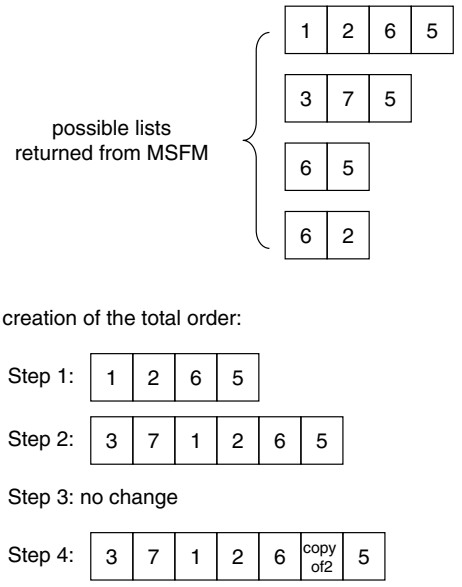

Fig. 16. Creating a total order between shared sets of transport level fields.

ordered set O to be equal to $\{1,2,6,5\}$. Adding the elements of $\{3,7,5\}$ into the set O results in inserting the elements 3 and 7 into O at position 1. The element 5 is not inserted since 5 is already an element of O. With the addition of 3 and 7 the set O becomes $\{3,7,1,2,6,5\}$. Similarly one can see that the elements of $\{6,5\}$ are not inserted into O because these elements already exist in O. Finally, the element 2 of the set $\{6,2\}$ is replicated. The total order between the sets of transport level fields returned from MSFM is $\{3,7,1,2,6,$ copy of $2,5\}$.

Our greedy algorithm creates a total order between all pointers returned from MSFM by replicating some sets of transport level fields in the TCAM unit. Once the total order O is created, the sets of transport level fields in the TCAM unit can be used for classification in the transport level dimensions without any additional priority resolution mechanism other than what is supported by the off-the-shelf TCAM.

5.4.5. Creating an index for TCAM entries

Creating a total order between the sets of transport level fields stored in the TCAM unit is not sufficient to make the TCAM solution work. The lists of pointers returned from MSFM need to be encoded into a small number of bits. These bits need to be appended into the key which is sent to the TCAM unit once MSFM returns a match. These bits are used for enabling the entries of the transport

level field sets returned from MSFM only and not other entries. This is important for having a functionally correct classifier. This problem can be formulated as follows: We have a collection of sets of transport level fields identified by pointers $L_1, L_2, \ldots, L_m$ and a number of ordered lists of pointers $O^i = \{L_1^i, L_2^i, \ldots\}$ returned from MSFM where L_j^i points to one of $\{L_1, \ldots, L_m\}$. We want to create an index $I(O^i)$ such that if this index is appended into a key sent to a TCAM, then this key enables only the entries of the sets of transport level fields which are elements of O^i.

Our approach to solve the problem is the following. Since each set needs to be enabled by an index, each set must specify some relevant bits in the index and some values for these relevant bits. All the entries in the set must specify the same relevant bits and values. The naïve approach to produce the index would be to allocate a separate bit for every set in the index. This approach, however, would easily make the length of an index exceed the length of an entry of a commercial TCAM. We believe that we can do better. Since each value of the index needs to enable multiple sets at the same time, the sets which are simultaneously enabled need to specify values at *different* bits of the key. For this reason any two sets identified by pointers L_i, L_j must specify values at different bits of the index if and only if there exists an ordered list of pointers returned from MSFM that includes both L_i and L_j. If any two pointers L_i and L_j are not found in the same ordered list, then their corresponding sets of transport level fields can specify different values at the same bits of the key. This can happen because there is no ordered list of pointers that enables both of these sets simultaneously.

Because of these observations, the bits of the index can be divided into regions $r_1, r_2, \ldots, r_l$. In each region r_i a number of values equal to $|r_i|$ is specified, each one for a separate set of transport level fields plus a 'don't enable' value. By $|r_i|$ we mean the cardinality of the set of values specified in the region r_i. Each set of transport level fields specifies a value in one region only. The entries of the set mark the bits of this region as relevant and the bits of all the other regions as irrelevant. However, more than one set may specify values in the same region of the index as mentioned above. The total number of bits needed to represent the index is:

$$S = \sum_i \lceil \log_2 |r_i| \rceil. \tag{13}$$

The lists of pointers $O_1, O_2, \ldots$ returned from MSFM can be used for constructing an $m \times m$ matrix $\boldsymbol{R}$, where m is the maximum number of sets of transport level fields in a total order. The matrix $\boldsymbol{R}$ is constructed in the following manner: $\boldsymbol{R}_{ij} = 1$, if an ordered list O exists that includes both pointers L_i, L_j. Otherwise $\boldsymbol{R}_{ij} = 0$. Therefore the index creation problem can be transformed into the following mathematical problem: Given a set of elements $L_1, L_2, \ldots, L_m$ and an $m \times m$ matrix $\boldsymbol{R}$ we want to partition the set into subsets $I_1, I_2, \ldots$ of cardinalities $|I_1|, |I_2|, \ldots$ so that the following conditions hold: First, for any pair L_i, L_j for which $\boldsymbol{R}_{ij} = 1$, the elements L_i and L_j are not grouped in the same set. Second, the sum S defined above is minimized, where

$$|r_i| = |I_i| + 1. \tag{14}$$

The plus one in the identity above corresponds to the 'don't enable' value specified for the region r_i.

To solve this problem we use a heuristic algorithm that calculates sets $I_1, I_2, \ldots$ by taking into account that many ordered lists include the same elements either in their beginning or in their end. The elements which are in common usually represent sets of transport level fields extracted from partially-specified filters. Such filters describe policies that apply to all traffic associated with a source or destination (e.g., forward all packets associated with established TCP connections). While ordered lists have some elements in common, ordered lists do not contain a large number of elements in average. For example the maximum size of each ordered list returned from MSFM is 11, in the databases we experimented with as mentioned earlier. These observations motivate the design of a greedy heuristic algorithm for calculating sets $I_1, I_2, \ldots$ which operates as follows:

Initially, the algorithm determines which elements are common to all ordered lists. These elements are removed from the ordered lists and are not placed in the partition $\{I_1, I_2, \ldots\}$. These elements correspond to sets of transport level fields which are always enabled when a match is found by MSFM. Then, the algorithm sets the partition $\{I_1, I_2, \ldots\}$ equal to $\{\{L_1^i\}, \{L_2^i\}, \ldots\}$ where L_1^i, L_2^i are the elements of the ordered list $O^i = \{L_1^i, L_2^i, L_3^i, \ldots, L_m^i\}$ that has the maximum number of elements. Next, the algorithm sets a variable called *insert-index* equal to 1. Subsequently, the algorithm determines the ordered list $O^j = \{L_1^j, L_2^j, \ldots\}$ that has the maximum number of new

elements (i.e., elements that have not been included in any partition before). The partition $\{I_1, I_2, \ldots\}$ is updated as discussed below: Let k be the smallest index greater than or equal to *insert-index* such that I_k contains L_1^j. If k exists the algorithm sets *insert-index* equal to $k+1$. Else the algorithm checks if the set I_k contains at least one element that cannot be represented in the same region of the index as L_1^j. If I_k does not contain any such element, then the algorithm inserts the element L_1^j in the set I_k and increments the variable *insert-index* by one. If L_1^j cannot be inserted in I_k then the algorithm determines the smallest index $l \geqslant k$ of a set I_l where the element L_1^j can be inserted. The algorithm inserts L_1^j in the set I_l and sets the index *insert-index* variable equal to $l+1$. The previous step is repeated for all elements of the ordered list O^j and for all other ordered lists until all elements have been inserted in the partition.

An example of the operation of our heuristic algorithm is illustrated in Fig. 17. This example of Fig. 17 follows the example of Fig. 16. In the example, the initial partition is {1}, {2}, {6}, {5}. The addition of the ordered list {3,7,5} changes the partition to {1,3}, {2,7}, {6}, {5}. In the last step of the algorithm the ordered list {6, copy of 2} is added and the partition becomes {1,3}, {2,7}, {6}, {5, copy of 2}. To understand why such a partition can be used for creating an index that enables the correct transport level field entries in the TCAM, let us consider what happens when MSFM returns the ordered list {3,7,5}. The index associated with this ordered list specifies the value for the set 3 in the first region of the index, the value for the set 7 in the second region of the index, the 'don't enable' value in the third region (to skip the transport level field set 6) and the value for 5 in the last index region. In this way the correct TCAM entries are enabled.

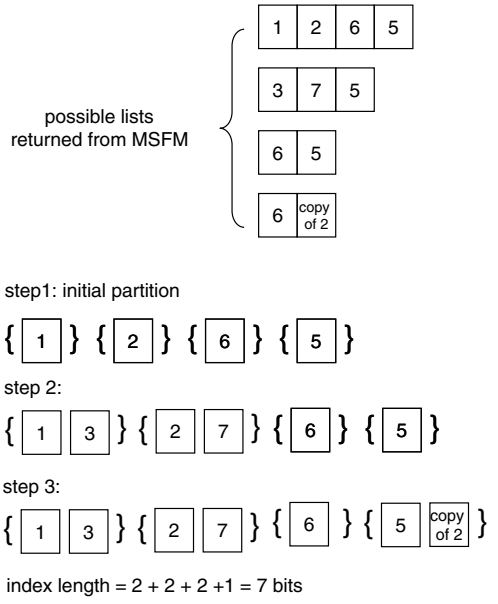

Fig. 17. Creating an index for TCAM entries.

The intuition behind this algorithm is that when we add the elements of a new ordered list into the partition, some elements in the new ordered list are likely to be found in the partition already. Therefore we do not to add these elements again. In addition for every new element, there is great likelihood that we can find an existing set in the partition where the new element can be inserted. All the elements in this set are represented by the same bits in the index. Hence, it is expected that the total number of bits needed to represent the sets of the partition (which is equal to the sum S above) is not going to be as large as the number of elements. For the access control lists we experimented with the length of the index ranged from 9 to 44 bits.

6. Evaluation

To evaluate our approach we compared it against a number of well known classification algorithms described in the literature. Comparing algorithm implementations is difficult since the performance of implementations varies depending on how implementations are optimized. For the schemes presented here, we have either used source code available in the public domain or re-implemented the schemes as described in their respective publications. Our implementations were done the 'straightforward way'. Even though our results may not demonstrate the most optimal performance for some of the state-of-the-art algorithms they can be used for qualitative comparison showing the basic property of our approach which is classification in predictable number of steps with reasonable memory requirement.

From among the parallel bit vector algorithms we implemented the Bit Vector (BV) and Aggregate Bit Vector (ABV) techniques described in [4,13] respectively. From among the heuristic algorithms we implemented the Recursive Flow Classification algorithm supporting the system configurations described in [7]. From among the hash-based algorithms we implemented the Tuple Space Search

algorithm described in [9]. From among the family of multidimensional trie algorithms we used the extended Grid-of-Tries (EGT) implementation found in [18]. From among the hierarchical cutting schemes we implemented the HiCuts [8] and HyperCuts [17] schemes.

We evaluated the combination of the improved most specific filter matching (MSFM) scheme and the Transport Level Sharing (TLS) scheme, implemented in four different ways: (i) without hardware acceleration (ii) with a custom hardware unit consisting of a single bin (iii) with a custom hardware unit consisting of multiple bins (iv) using an off-the-shelf TCAM unit.

Our evaluation results are presented in Tables 9–11. Table 9 shows the memory requirements of the classification schemes considered. Table 10 shows the number of dependent memory accesses involved in the lookup process of each algorithm. By 'memory accesses' in this context we mean dependent read or write operations that can fetch or write one or multiple words at the same time to/from memory. The number of memory accesses is not the same as the number of words read during the lookup process of an algorithm. The number of memory accesses is simply the number of dependent steps involved in the lookup process. Table 11 shows the memory access bandwidth requirement of each algorithm. By memory access bandwidth we mean the maximum number of words that need to be fetched or written in a single memory access or dependent step of an algorithm.

Table 9
Lookup time comparison, expressed in worst case dependent memory accesses

	Number of rules	TSS	RFC
ACL 1	754	83	10
ACL 2	605	200	10
ACL 3	2399	361	10
ACL 4	157	18	10
		ABV	EGT
ACL 1	754	10	84
ACL 2	605	10	136
ACL 3	2399	13	150
ACL 4	157	9	108
		HiCuts	HyperCuts
ACL 1	754	28	15
ACL 2	605	33	19
ACL 3	2399	37	25
ACL 4	157	25	14
		MSFM/TLS, without HW acceleration	MSFM/TLS, with a single bin
ACL 1	754	11	11
ACL 2	605	11	11
ACL 3	2399	11	11
ACL 4	157	11	11
		MSFM/TLS, with multiple bins	MSFM/TLS, with a TCAM
ACL 1	754	11	11
ACL 2	605	11	11
ACL 3	2399	11	11
ACL 4	157	11	11

Table 10
Memory requirement (expressed in KB)

	Number of rules	TSS	RFC
ACL 1	754	249	1274
ACL 2	605	600	562
ACL 3	2399	1058	2044
ACL 4	157	54	367
		ABV	EGT
ACL 1	754	510	183
ACL 2	605	266	188
ACL 3	2399	2988	538
ACL 4	157	12	22
		HiCuts	HyperCuts
ACL 1	754	42	38
ACL 2	605	253	40
ACL 3	2399	635	65
ACL 4	157	13	4
		MSFM/TLS, without HW acceleration	MSFM/TLS, with a single bin
ACL 1	754	58	58
ACL 2	605	175	175
ACL 3	2399	446	446
ACL 4	157	19	19
		MSFM/TLS, with multiple bins	MSFM/TLS, with a TCAM
ACL 1	754	31	29
ACL 2	605	74	67
ACL 3	2399	213	197
ACL 4	157	6	5

Table 10 indicates that our scheme requires no more than 11 sequential memory accesses independent of the size of the database in all possible implementations. The number 11 comes from the number of dependent steps required for traversing two 4-bit trie data structures in parallel (8) plus one more dependent step for accessing the tables of MSFM plus two more dependent steps for classifying packets in the transport level field dimensions. Classification on the transport level field dimensions depends on the flavor of MSFM/TLS which is being imple-

Table 11
Memory access bandwidth comparison, expressed in worst case words fetched/written per accesses

	Number of rules	TSS	RFC
ACL 1	754	1	1
ACL 2	605	1	1
ACL 3	2399	1	1
ACL 4	157	1	1
		ABV	EGT
ACL 1	754	1	1
ACL 2	605	1	1
ACL 3	2399	1	1
ACL 4	157	1	1
		HiCuts	HyperCuts
ACL 1	754	1	1
ACL 2	605	1	1
ACL 3	2399	1	1
ACL 4	157	1	1
		MSFM/TLS, without HW acceleration	MSFM/TLS, with a single bin
ACL 1	754	64	64
ACL 2	605	64	64
ACL 3	2399	64	64
ACL 4	157	64	64
		MSFM/TLS, with multiple bins	MSFM/TLS, with a TCAM
ACL 1	754	8	4
ACL 2	605	8	4
ACL 3	2399	8	4
ACL 4	157	8	4

mented. In the case of MSFM without hardware acceleration this step returns the values of the port number ranges and protocol fields used in the second stage of the packet classification process. This information if appropriately compressed (e.g., source port numbers are almost always equal to the wildcard) can be represented using at most 64 words. This information is processed in software. In the case of MSFM with singe bin hardware acceleration this step also returns the same information. However, this time transport level field values are sent to a hardware accelerator instead of being processed in software. In the case of MSFM with multiple bins the information sent to the hardware accelerator is a list of bin pointers. This information is at most 8 words. Finally in the case of MSFM with TCAM-based acceleration the information sent to the TCAM unit is a 4 word key.

In our implementation we avoid storing transport level field information (either field values or pointers) inside the MSFM tables, in order to keep their size small. MSFM tables return just a pointer to some memory area where transport level field information can be obtained. One can see that in all flavors of MSFM/TLS transport level field classification requires two dependent steps, hence the number 11. What differentiates implementations is their memory access bandwidth requirement. The software-only implementation requires 8 times more memory access bandwidth than the flavor of MSFM/TLS with multiple bins and 16 times more memory access bandwidth than the flavor of MSFM/TLS with a TCAM. The most optimal implementation of MSFM/TLS is with an off-the-shelf TCAM even though this implementation requires extra TCAM space for storing transport level fields.

The only other scheme which also appears to be associated with constant time requirement is RFC. Unlike RFC however, our scheme avoids significant increases in the lookup data structure size because many cross products are removed form the lookup table and because the unique instances of transport level field entries, which are a few, are stored only once in a specialized hardware unit.

One of the fastest alternative packet classification schemes shown in the tables is HyperCuts [17]. It is shown in Tables 9 and 10 that HyperCuts requires a variable number of dependent steps (i.e., 14–25) for classifying packets from our sample databases while demonstrating smaller memory requirement than the combination of MSFM and TLS. The time requirement of multidimensional cutting scheme depends on the classification database used. This happens because multidimensional cutting builds a decision tree of varying height and because the number of rules which are stored in each leaf node of the tree varies as well. In contrast, MSFM uses parallel tries of fixed worst case height. As a result, MSFM requires the same predictable number of steps for classifying packets for any database used. On the other hand one can observe that HyperCuts requires approximately 3–8 times less memory than MSFM/TLS which is because HyperCuts does not need to pay the penalty of storing indicator filters. This is one of the main advantages of HyperCuts as compared to our approach. The reader should also note that our HiCuts and HyperCuts implementations perform a linear search on the list of rules found after a tree search is complete as described in their respective papers [8,17]. These implementations can be optimized reading all rule data from memory in parallel as our MSFM/TLS implementation does.

Other optimizations can be thought of that combine multidimensional cutting in the source–destination IP address dimensions and transport level sharing in the remaining dimensions. Such optimizations however are left for future work.

To verify the scalability of our approach in terms of supporting constant worst case time requirement we built a synthetic ACL generator. Using this generator we created synthetic databases of sizes ranging from 4K to 10K rules satisfying the IP prefix pair and transport level sharing properties reported in the literature and discussed in this paper. Our synthetic ACL generator satisfies many of the properties of the rules characterizing large classification databases including (i) the breakdown of filters between partially- and fully-specified observed in [16]; (ii) the breakdown of fully-specified filters between domain–host, host–domain, host–host and domain–domain filters observed in [16]; (iii) the breakdown of partially-specified filters between filters specifying the wildcard in the source and destination addresses characterizing the databases we experimented with; (iv) the filter overlap properties reported in [16]; (v) the transport level field sharing properties reported in Section 3; and (vi) the structure of the entries of the unique sets of transport level fields reported in [16].

The ACL generation algorithm is outlined as follows: First, the algorithm determines the number of fully- and partially-specified filters in the new synthetic database, the number of unique sets of transport level fields, and the number of unique entries in the sets of transport level fields in the database. The algorithm determines these numbers from the requested number of rules in the new database and from the observed levels of sharing found in real databases. Second, the entries of the unique sets of transport level fields are generated obeying the properties of protocol fields and port ranges reported in the literature. Third, the unique sets of transport level fields are created from synthetically generated entries obeying the size distributions observed in the literature and in this paper. Next partially- and fully-specified filters are generated satisfying the overlap properties that typically characterize large ISP databases. Finally the ACL is constructed by combining the generated filters with the transport level fields.

We observed that the synthetic databases, like the real databases were associated with a predictable number memory accesses in the critical path. The memory requirement for synthetic databases was between 400 KB and 3.8 MB. The properties of a synthetic database of 11,526 rules are shown in Table 12.

Our TCAM-based implementation of the Transport Level Sharing scheme is further analyzed in Table 13. We observe from Table 13 that the amount of TCAM entries which are required for storing the unique instances of sets of transport level fields in a database is significantly smaller than the number of entries required for storing the rules of the entire database. A TCAM that stores the unique sets of the transport level fields of the databases we experimented with requires 21.6–65.7% of the entries of a hardware unit that stores the transport level fields of all rules in the same databases. We define the 'TCAM compression factor' as the ratio between the number of TCAM entries required for storing all the rules in a database over the number of TCAM entries required for storing the shared sets of transport level fields in the database.

Fig. 18 illustrates the TCAM compression factor as a function of the database size for the access control lists we experimented with. Fig. 13 shows that the larger a database is the larger TCAM compression factor the database is associated with. The intuition behind this observation is that the number of

Table 12
Properties of a synthetic database of 11,526 rules

Property	Value
Number of rules	11,526
Number of unique IP prefix pairs	5824
Number of unique source prefixes	3187
Number of unique destination prefixes	3560
Number of unique $(X, *)$ partially-specified filters	220
Number of unique $(*, Y)$ partially-specified filters	506
Number of shared sets of transport level fields	682
Number of entries in unique sets of transport level fields	1602
Number of bits read at a time when traversing the tries	16-8-4-4
Number of 16 entry unique source trie blocks	3102
Number of 16 entry unique destination trie blocks	5174
Number of 256 entry unique source trie blocks	314
Number of 256 entry unique destination trie blocks	434
Number of 64K entry unique source trie blocks	1
Number of 64K entry unique destination trie blocks	1
Number of unique partial IP prefix pair overlaps	46,028
Number of unique indicator filters	259,794
Type of HW acceleration	Multiple bin
Number of worst case dependent memory accesses	7
Space requirement (MB)	3.8

Table 13
Analysis of a TCAM-based implementation

	Number of TCAM entries needed to store the unique sets of transport level fields	Number of TCAM entries needed to store all the rules in the database
ACL 1	1210	1843
ACL 2	336	1243
ACL 3	947	4387
ACL 4	90	156

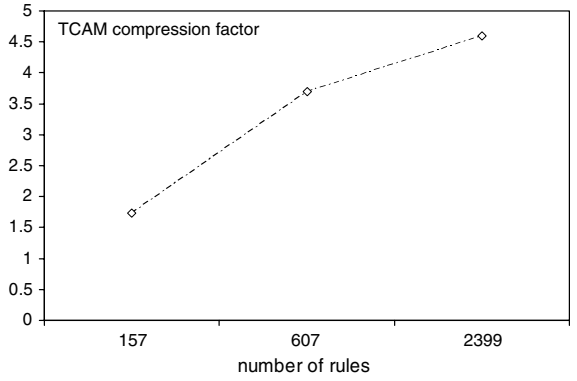

Fig. 18. TCAM compression factor as a function of the ACL.

source–destination IP prefix pairs increases at much higher rate than the number of transport level fields as the size of the database increases.

The main penalty we pay to support both predictable lookup time and reduced data structure size is the complexity of the update process. The update process is currently time consuming since it requires the creation of lookup tables, the reordering of rules, the discovery of shared sets of transport level fields and the creation of the data structures used by the hardware accelerator. Some of the data structures used by the update process are dynamically created during by the classifier when bursts of new rules are added and hence they do not need to be maintained over the course of the system's operation. The additional data structures which are created dynamically include a conflict resolution trie [19] in order to determine the unique overlaps between IP prefix pairs and modified versions of the tables of the lookup process in order to determine new indicator filters. The classifier modifies the tables of the lookup process during the update time in order to correctly determine the indicator filters associated with every new IP prefix pair. In these modified lookup tables each table entry does not only contain a pointer to transport level sharing data but also the indicator or most specific filter associated with the entry. Details about our update process are provided in the Appendix.

The largest worst case update latency was found to be 197 thousand memory accesses for adding a new rule into ACL3 and 103 million memory accesses for creating all lookup data structures of ACL3. These numbers indicate that our classifier is not suitable for highly dynamic networking environments where new rules need to be added into classifiers over intervals smaller than the millisecond time frame. Our update latency is also larger than the update latency of other well known classification schemes. In the case of HyperCuts for example, an update can be performed by adding a new rule into an existing search tree without recreating the tree. Thus the update time does not exceed the time to traverse the tree and the linear list of rules, which is 14–25 memory accesses (from Table 9). Similar to HyperCuts our scheme can be modified to support fast update times at the expense of the optimality of the lookup algorithm and data structure. A modified version of our scheme could find all possible matches for a packet in the source–destination IP address dimensions using any suitable tree structure and then use transport level sharing to perform classification in the remaining dimensions. Such optimization, however, is the subject of future work.

7. Concluding remarks

In this paper we described a hybrid scheme, where a parallel LPM lookup algorithm implemented in software determines the most specific filter for a packet and a specialized hardware unit determines if the packet matches any of the transport level fields of a database. The most significant contribution of our work is that our scheme can classify packets in a small and predictable number of steps which is independent of the number of rules in a database, while keeping its memory requirement at reasonable level. Unfortunately predictable time performance and reasonable memory requirement do not come for free. Our scheme requires hardware acceleration in order to operate efficiently. The additional hardware required for implementing our scheme is not as costly as a pure TCAM solution though. Another penalty we pay to support both predictable lookup time and reduced data structure size is the complexity of the update process. Our classification scheme is not be suitable for highly dynamic networking environments where

new rules need to be added into classifiers over intervals smaller than the millisecond time frame.

Despite these two drawbacks, we believe that our work has some importance because it presents one of few schemes that can reduce both the lookup time and space requirement of a classifier. We further believe that our observations on real world databases can help the research community gain a better understanding on the relationship between the properties and performance of packet classifiers. Our observations and system design are novel and can potentially pave the way for further innovations on the design of high performance packet classification systems for the future Internet.

Appendix

Proof of Theorem 1. To prove Theorem 1 it is suffice to show that any element of the sets of not covered cross products $F_{NC}(F)$ and partially covered cross products $F_{PC}(F)$ defined by (6) and (7) is not included into the set of fully covered cross products $F_{FC}(F)$ defined by (9). We also need to show that and that every element of the set of cross products $F_C(F)$ is either included into one of the sets $F_{NC}(F)$ and $F_{PC}(F)$ or $F_{FC}(F)$. The first of the statements is true because any not covered or partially covered cross product is neither an element of the set of fully-specified filters $F_F(F)$ nor the set of intersections that are fully-specified $F_F(F_I(F))$ nor it is contained into any element of these sets. The second statement is also true since the union of the sets of not covered cross products $F_{NC}(F)$, partially covered cross products $F_{PC}(F)$, and fully covered cross products $F_{FC}(F)$ is the set $F_C(F)$ of all cross products of F. □

Proof of Theorem 2. First we prove statement (i): If the most specific filter for a packet F_{MS} is a fully-specified filter and the stage of parallel LPM searches returns another filter $F_{LPM} = (S_{LPM}, D_{LPM})$ which completely contains the most specific filter F_{MS} (i.e., $F_{LPM} \neq F_{MS}$ and $F_{LPM} \prec F_{MS}$), then the source prefix S_{LPM} must be a prefix of S_{MS} or the destination prefix D_{LPM} must be a prefix of D_{MS}. Let us first assume that S_{LPM} is a prefix of S_{MS}. We can show that this is not possible since both S_{LPM} and S_{MS} match the source IP address of the packet. In this case the LPM search on the source IP address dimension would return the prefix S_{MS} of the most specific filter and not S_{LPM}. If D_{LPM} is a prefix of D_{MS}, we can also show that this is not possible since both prefixes D_{LPM} and D_{MS} match the destination IP address of the packet. In this case the LPM search on the destination IP address dimension would return the prefix D_{MS} and not D_{LPM}. Hence statement (i) is proven.

To prove statement (ii) we consider the case when the most specific filter for a packet $F_{MS} = (S_{MS}, D_{MS})$ is a partially-specified filter for which $S_{MS} \neq * \wedge D_{MS} = *$. If the LPM search on the source IP address dimension returns a prefix S_{LPM} which is different from S_{MS}, then S_{LPM} cannot be a prefix of S_{MS} since S_{LPM} is returned from an LPM search. On the other hand, if the prefix S_{MS} is a prefix of S_{LPM}, then this means that there exists a partially-specified filter equal to $(S_{LPM}, *)$ which covers the packet p and is contained into the most specific filter F_{MS}. However, this cannot be true because F_{MS} is the most specific filter for packet p. Hence, the prefix S_{LPM} returned from the stage of parallel LPM searches is equal to the source prefix of the most specific filter S_{MS}. The proof of statement (iii) is similar. □

Description of the update process: The update process consists of three steps. In the first step, a new rule is added into a set of rule-sets. All rules in a rule-set specify the same IP prefix pair. A rule may be added either into an existing rule-set or a new rule-set. If the rule is added to a new rule-set, then a second step takes place, where the source–destination IP prefix pair of the new rule is inserted into all classification data structures which store IP prefix pair information. If the rule is added into an existing rule-set then the second step is omitted. A third step is executed afterwards where the unique sets of transport level fields associated with the most specific filter of every packet are determined and added into the classifier.

Adding a new IP prefix pair

In what follows we provide a description of the process of adding a new IP prefix pair for the non-trivial case where the IP prefix pair is fully specified and thus creates fully-specified filter intersections with other filters. First, a search is made on a source trie data structure. If the source prefix of the new rule is not included into the source trie, then this prefix is inserted into the trie. A new index is assigned to this prefix, in this case. Otherwise the index associated with the source prefix is obtained

from the trie. A similar procedure is followed for the destination IP prefix of the new rule.

Once the source and destination IP prefixes of the new rule are inserted into the source and destination trie data structures, a search is made on the conflict resolution trie. The search returns all partial overlaps formed between the new filter and the filters in conflict resolution trie. These filters are added into conflict resolution trie.

Next, for each new filter added into the conflict resolution trie a set of indicator filters is determined. Indicator filters are determined by traversing the source and destination trie data structures and by obtaining all prefixes that match with the source and destination prefixes of every new filter added into the conflict resolution trie. The cross products determined by these source and destination prefixes are candidate indicator filters to be added into the database.

For every candidate indicator filter a lookup is performed on the primary table. The entry associated with the indicator filter is determined. If this entry is empty then the indicator filter is added into the primary table. If the filter has already been added into the primary table, then the filter is not added twice. In case of a collision the hash table is updated accordingly. In the case of a wide filter no insertions into the primary table are made but the wide filter is inserted into a separate Cross Producting table.

Creating sets of transport level fields

In the third step of the update process unique sets of transport level fields are created. A total order is created between these shared sets as discussed in Section 5 (for the TCAM-based flavor). Once the total order is created, an index is determined for each ordered list of pointers returned from MSFM as discussed in Section 5. The MSFM data structures are updated accordingly and the shared sets of transport level fields are inserted into the TCAM.

References

[1] P. Gupta, N. McKeown, Algorithms for packet classification, IEEE Network Magazine (March/April) (2001).

[2] P. Tsuchiya, A search algorithm for table entries with non-contiguous wildcarding, Technical Report, Bellcore.

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

[4] T.V. Lakshman, D. Stiliadis, High-speed policy-based packet forwarding using efficient multi-dimensional range matching, in: Proceedings of ACM SIGCOMM, 1998.

[5] M.M. Buddhikot, S. Suri, M. Waldvogel, Space decomposition techniques for fast layer-4 switching, in: Proceedings of the Conference on Protocols for High Speed Networks, 1999.

[6] A. Feldman, S. Muthukrishnan, Tradeoffs for packet classification, in: Proceedings of IEEE INFOCOM, 2000.

[7] P. Gupta, N. McKeown, Packet classification on multiple fields, in: Proceedings of ACM SIGCOMM, 1999.

[8] P. Gupta, N. McKeown, Packet classification using hierarchical intelligent cuttings, IEEE Micro (January/February) (2000).

[9] V. Srinivasan, S. Suri, G. Varghese, Packet classification using tuple space search, in: Proceedings of ACM SIGCOMM, 1999.

[10] F. Shafai, K.J. Schultz, G.F.R. Gibson, A.G. Bluschke, D.E. Somppi, Fully parallel 30-MHz, 2.5 Mb CAM, IEEE Journal of Solid-State Circuits (November) (1998).

[11] P. Warkhebe, S. Suri, G. Varghese, Fast packet classification for two-dimensional conflict free filters, in: Proceedings of IEEE INFOCOM, 2001.

[12] F. Baboescu, S. Singh, G. Varghese, Packet classification for core routers: is there an alternative to CAMs? in: Proceedings of IEEE INFOCOM, 2003.

[13] F. Baboescu, G. Varghese, Scalable packet classification, in: Proceedings of ACM SIGCOMM, 2001.

[14] M. Degermark, A. Brodnik, S. Carlsson, St. Pink, Small forwarding tables for fast routing lookups, in: Proceedings of ACM SIGCOMM, 1997.

[15] D. Taylor, J. Turner, Scalable packet classification using distributed Cross Producting of field labels, Poster Session of ACM SIGCOMM, 2004.

[16] M.E. Kounavis, A. Kumar, H. Vin, R. Yavatkar, A.T. Campbell, Directions in packet classification for network processors, in: Proceedings of the Second Workshop on Network Processors, 2003.

[17] S. Singh, F. Baboescu, G. Varghese, J. Wang, Packet classification using multidimensional cutting, in: Proceedings of ACM SIGCOMM, 2003.

[18] S. Singh, Packet Classification Repository, Public Domain Source Code, University of California, San Diego, 2003.

[19] A. Hari, S. Suri, G. Parulkar, Detecting and resolving packet filter conflicts, in: Proceedings of IEEE INFOCOM, 2000.

[20] M.E. Kounavis, A. Kumar, H. Vin, R. Yavatkar, Method and apparatus for two stage packet classification using most specific filter matching and transport level sharing, United States Patent Application, No. 20050083935, Filed in 2003.

[21] A. Kumar, M.E. Kounavis, R. Yavatkar, H. Vin, P. Chandra, S. Lakshmanamurthy, C. Kuo, Apparatus and method for two stage packet classification using most specific filter matching and transport level sharing, United States Patent Application, No. 20050226235, Filed in 2004.

[22] M.E. Kounavis, A. Kumar, R. Yavatkar, H. Vin, Line rate packet classification and scheduling, in: Tutorial, First Symposium on Architectures for Networking and Communication Systems (ANCS 2005), Princeton, NJ, October 2005.

Michael Kounavis is a senior Research Scientist at Intel Corporation. He joined Intel in 2003. Since then, he has worked on developing novel algorithms and solutions for line rate packet processing and data integrity. His current research focuses on accelerating cryptographic algorithms and protocols. Prior to joining Intel he worked on programming network architectures at Columbia University. He has published over 30 papers in leading technical journals and conferences and he is a program committee member of the IEEE ISCC, SPECTS and IWDDS conferences.

Alok Kumar is a staff Software Engineer in the Digital Enterprise Group at Intel Corporation. His interests are in the areas of high-speed programmable routers, quality of service, and computer graphics. He received his B.Tech degree in Computer Science from the Indian Institute of Technology, Delhi, in 1999, and his M.S. degree in Computer Science from the University of Texas at Austin in 2001.

Raj Yavatkar is an Intel Fellow and Director of the Platform Validation Architecture in the Digital Enterprise Group. Previously, he led the formation of the Systems Technology Lab involved in advanced R&D in the areas of system architecture and platform technologies. From 1999 through 2004, he was the Chief Software Architect for Intel's IXP family of network processors. At Intel, he led Intel's advanced research and development activities in internet quality of service and programmable networks. He designed a framework for policy-based network management that led to development of an industry-wide technical standard. He received his Ph.D. in Computer Science from Purdue University in 1989 and holds fourteen patents, with more than 20 pending. He is recognized as a leading expert in the networking industry, and was the General Chair of ACM Sigcomm 2004. He has authored or co-authored five Internet standards. He has also published more than 40 papers in academic journals and conferences and has co-authored the book, Inside the Internet's Resource Reservation Protocol (RSVP) published by John Wiley. He serves on the editorial board of the IEEE Network magazine and previously served as an editor of Computer Communications, ACM/Springer-Verlag Journal on Multimedia Systems and Kluwer's Multimedia Tools and Applications.

Harrick Vin is a Professor of Computer Sciences at the University of Texas at Austin. His research interests are in the areas of networks, operating systems, distributed systems, and multimedia systems. He received his Ph.D. in Computer Science from the University of California at San Diego in 1993. He has co-authored more than 100 papers in leading journals and conferences. He is a recipient of several awards including the Faculty Fellow in Computer Sciences, Dean's Fellowship, National Science Foundation CAREER award, IBM Faculty Development Award, Fellow of the IBM Austin Center for Advanced Studies, AT&T Foundation Award, National Science Foundation Research Initiation Award, IBM Doctoral Fellowship, NCR Innovation Award, and San Diego Supercomputer Center Creative Computing Award. He has served on the Editorial Board of ACM/Springer Multimedia Systems Journal, IEEE Transactions on Multimedia, and IEEE Multimedia. He has been a guest editor for IEEE Network. He has served as the program chair, the program co-chair, and a program committee member for several conferences.