Report Number: WUCSE-2003-65

2003-05-16

Compressed Data Structures for Recursive Flow Classification

Edward W. Spitznagel

High-speed packet classification is crucial to the implementation of several advanced network services and protocols; many QoS implementations, active networking platforms, and security devices (such as firewalls and intrusion-detection systems) require it. But performing classification on multiple fields, at the speed of modern networks, is known to be a difficult problem. The Recursive Flow Classification (RFC) algorithm described by Gupta and McKeown performs classification very quickly, but can require excessive storage when using thousands of rules. This paper studies a compressed representation for the tables used in RFC, trading some memory accesses for space. The compression's efficiency can be... **Read complete abstract on page 2.**

Compressed Data Structures for Recursive Flow Classification

Edward W. Spitznagel

Complete Abstract:

High-speed packet classification is crucial to the implementation of several advanced network services and protocols; many QoS implementations, active networking platforms, and security devices (such as firewalls and intrusion-detection systems) require it. But performing classification on multiple fields, at the speed of modern networks, is known to be a difficult problem. The Recursive Flow Classification (RFC) algorithm described by Gupta and McKeown performs classification very quickly, but can require excessive storage when using thousands of rules. This paper studies a compressed representation for the tables used in RFC, trading some memory accesses for space. The compression's efficiency can be improved by rear-ranging rows and columns of the tables. Finding a near-optimal rearrangement can be transformed into a Traveling Salesman Problem in which certain approximation algorithms can be used. Also, in evaluating the compressed representation of tables, we study the effects of choosing different reduction trees in RFC. We evaluate these methods using a real-world filter database with 159 rules. Results show a reduction in the size of the cross product tables by 61.6% in the median case; in some cases their size is reduced by 87% or more. Furthermore, experimental evidence suggests larger databases may be more compressible.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2003-65

Compressed Data Structures for Recursive Flow Classification

Authors: Spitznagel, Edward W.

May 16, 2003

Abstract: High-speed packet classification is crucial to the implementation of several advanced network services and protocols; many QoS implementations, active networking platforms, and security devices (such as firewalls and intrusion-detection systems) require it. But performing classification on multiple fields, at the speed of modern networks, is known to be a difficult problem.

The Recursive Flow Classification (RFC) algorithm described by Gupta and McKeown performs classification very quickly, but can require excessive storage when using thousands of rules. This paper studies a compressed representation for the tables used in RFC, trading some memory accesses for space. The compression's efficiency can be improved by rearranging rows and columns of the tables. Finding a near-optimal rearrangement can be transformed into a Traveling Salesman Problem in which certain approximation algorithms can be used. Also, in evaluating the compressed representation of tables, we study the effects of choosing different reduction trees in RFC.

We evaluate these methods using a real-world filter database with 159 rules. Results show a reduction in size of the crossproduct tables by 61.6% in the median case; in some cases their size is reduced by 87% or more. Furthermore, experimental evidence suggests larger databases may be more compressible

Compressed Data Structures for Recursive Flow Classification

Edward W. Spitznagel

WUCSE-2003-65

16 May 2003

Department of Computer Science and Engineering
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

High-speed packet classification is crucial to the implementation of several advanced network services and protocols; many QoS implementations, active networking platforms, and security devices (such as firewalls and intrusion-detection systems) require it. But performing classification on multiple fields, at the speed of modern networks, is known to be a difficult problem.

The Recursive Flow Classification (RFC) algorithm described by Gupta and McKeown performs classification very quickly, but can require excessive storage when using thousands of rules. This paper studies a compressed representation for the tables used in RFC, trading some memory accesses for space. The compression's efficiency can be improved by rearranging rows and columns of the tables. Finding a near-optimal rearrangement can be transformed into a Traveling Salesman Problem in which certain approximation algorithms can be used. Also, in evaluating the compressed representation of tables, we study the effects of choosing different reduction trees in RFC.

We evaluate these methods using a real-world filter database with 159 rules. Results show a reduction in the size of the crossproduct tables by 61.6% in the median case; in some cases their size is reduced by 87% or more. Furthermore, experimental evidence suggests larger databases may be more compressible.

Compressed Data Structures for Recursive Flow Classification

Edward W. Spitznagel
ews1@cse.wustl.edu
Washington University
St. Louis, MO 63130-4899
This work was supported by the National Science Foundation, ANI-9813723.

1. Introduction

Packet classification is important for a multitude of emerging network services. Advanced network services such as DiffServ edge routers [12], firewalls, intrusion-detection devices, and many QoS-enabled routers need to classify packets to determine what to do with them. But it is difficult to perform general packet classification at modern network backbone speeds (e.g. 40Gb/s and up.)

Internet routing lookups, in the past, were based only on finding the longest matching prefix for a packet's destination address. This is essentially packet classification in one dimension, a process for which several algorithms are known [3] [11] [16] [8] [13] [5].

Packet classification in two dimensions (e.g. classifying by Source Address and Destination Address) is more difficult, but reasonable algorithms for this case exist also [10] [14] [4] [6]. These algorithms, however, do not scale well enough for general use with more than two dimensions.

In the most general case, classifying packets in K dimensions is provably hard for K greater than 2 [10] [14] [15] [7] [9]. It has been shown to require $O(log^{K-1}N)$ time and linear space, or $logN$ time and $O(N^K)$ space, where N is the number of rules [10].

Ternary Content-Addressable Memories (TCAMs) are circuits that essentially compare a packet's headers against every rule in parallel [12]. TCAMs are fast, but expensive; they also scale poorly due to power dissipation and board space issues.

Fortunately, the rules in real-world classifiers tend to exhibit certain properties; these properties allow clever algorithms to beat the worst-case bounds in most real-world applications [15] [7] [9] [2]. Thus, there have been efforts to develop classification algorithms that perform well in these "typical" cases:

Recursive Flow Classification (RFC) [7] appears to be the fastest packet classification algorithm in current literature; HiCuts [9], ABV [1], and EGT [2] are not quite as fast, but typically require considerably less memory. A repository containing papers and source code for some of these algorithms has been established [17].

The RFC algorithm described by P. Gupta and N. McKeown has excellent performance in terms of time to classify a packet, but with a large filter database the storage requirements can become large. In this paper, we describe and evaluate methods for improving the storage efficiency of the RFC scheme.

1.1. Packet Classification Problem

The object of packet classification is to categorize packets by applying a set of rules called *filters* to the header fields of a packet. Each rule consists of a specification of header field values, and an action to perform on packets whose headers match that specification.

The information relevant for classifying a packet is contained inside the packet in K distinct header fields, denoted $H[1], H[2], ..., H[K]$. For example, the fields typically used to classify Internet Protocol (IP) packets are the destination IP address, source IP address, destination port number, source port number, protocol number and protocol flags. The number of protocol flags is limited, so they are often combined into the protocol field itself.

Using those fields for classifying IP packets, a filter $F = (128.252.*, *, TCP, 23, *)$, for example, specifies a rule matching traffic addressed to subnet 128.252 using TCP destination port 23, which is used for incoming Telnet; using a filter like this, a firewall may disallow Telnet into its network.

A filter database consists of N filters $F_1, F_2, ..., F_N$. Each filter F_j is an array of K values, where $F_j[i]$ is a specification on the i-th header field. The i-th header field is sometimes referred to as the i-th dimension or the i-th axis, when considering a packet's header as specifying a point in K-dimensional space. The value $F_j[i]$ specifies what the i-th header field of a packet must contain in order for the packet to match filter j. These specifications often have (but need not be restricted to) the following forms: exact match, for example "source address must equal 128.252.169.16"; prefix match, like "destination address must match prefix 128.252.*"; or range match, e.g. "destination port must be in the range 0 to 1023."

Each filter F_j has an associated directive $disp_j$, which specifies the action to perform for a packet that matches this filter. This directive may indicate whether to block the packet, send it out a particular interface, or perform some other action. Filter databases look like the example in Table 1, but most real-world databases have many more filters in them.

A packet P is said to *match* a filter F if each field of P matches the corresponding field of F. For instance, let $F = (128.252.*, *, TCP, 23, *)$ be a filter with $disp = block$. Then, a packet with header (128.252.169.16, 128.111.41.101, TCP, 23, 1025) matches F, and is therefore blocked. The packet (128.252.169.16, 128.111.41.101, TCP, 79, 1025), on the other hand, doesn't match F.

Since a packet may match multiple filters in the database, we associate a *cost* for each filter to resolve ambiguous matches. The packet classification problem is to find the lowest cost filter matching a given packet P.

Destination Address	*Source Address*	*Dest. Port*	*Src. Port*	*Protocol and flags*	*comments comments*
host M_1	*	25	*	TCP	allow inbound mail to M_1
host M_2	*	53	*	UDP	allow DNS access to M_2
*	network N	*	*	*	allow outgoing packets
network N	*	*	*	TCP-ack	return ACKs OK
*	*	*	*	*	block everything else

Table 1: Example: simplified firewall filter database

To classify a packet, one could simply apply each rule, in increasing order of cost, until a match is found. This approach is easy to use, but often is not fast enough when a large number of rules are used. Several more sophisticated algorithms have been developed that use data structures cleverly to improve the speed of packet classification. Each algorithm's performance can be measured in terms of the time required to classify a packet, the storage space required for the algorithm's data

structures, and the complexity of updating the data structures when a filter is added, deleted or changed.

One particularly interesting packet classification algorithm is the Recursive Flow Classification (RFC) method described by Pankaj Gupta and Nick McKeown. RFC performs lookups in constant time (regardless of number of filters) but can have significant storage requirements. This paper examines methods for reducing the storage used in the RFC scheme.

In this paper, we describe and evaluate methods for improving the storage efficiency of RFC. Section 2 provides an explanation of the RFC algorithm itself. Section 3 examines possible methods for improving RFC, including a simple compression technique (Section 3.1), a heuristic for improving compression (Section 3.2), and the effect of varying the reduction tree structure used in the classifier (explained in Section 3.3). Following that are a few concluding remarks.

2. Recursive Flow Classification

This section provides a description of the Recursive Flow Classification algorithm. This algorithm works by processing the header fields of a packet in chunks. Effectively, the algorithm tracks which filters have been matched by the various header chunks, and combines the results for the chunks to determine the set of filters matched by the complete headers.

In order to do this efficiently, RFC uses two techniques described in detail later in this section. First, at each step it uses equivalence classes defined by the set of filters matched thus far in processing the packet; these equivalence classes are a concise way for the algorithm to keep track of which filters have been matched by the various chunks of the header fields. Secondly, in order to combine the results for different chunks together efficiently, RFC uses crossproducting tables to store precomputed results. Both of these techniques are described in the examples below, which build up from an overly simplified example to the basis of the full RFC algorithm.

2.1. Use of Equivalence Classes

Consider the following example, in which only one header field is used for classification; for simplicity, let this field represent a destination address of a mere four bits in length. The set of filters in this example are shown in Table 2.

Filter Number	*Destination Address (in binary)*	*Cost*
1	001*	1
2	0101-0111	2
3	110*	3
4	0001-1001	4

Table 2: Example 1-dimensional filter database

The filters in Table 2 can be projected graphically along an axis representing the domain of possible values for the destination address. The axis can be divided into intervals at the endpoints of each filter, as shown in Figure 1. Within each interval, a particular set of filters is matched.

We can use this to partition the set of possible values for this field (in this case, the space of all possible destination addresses) into equivalence sets, where all values in a set match exactly the

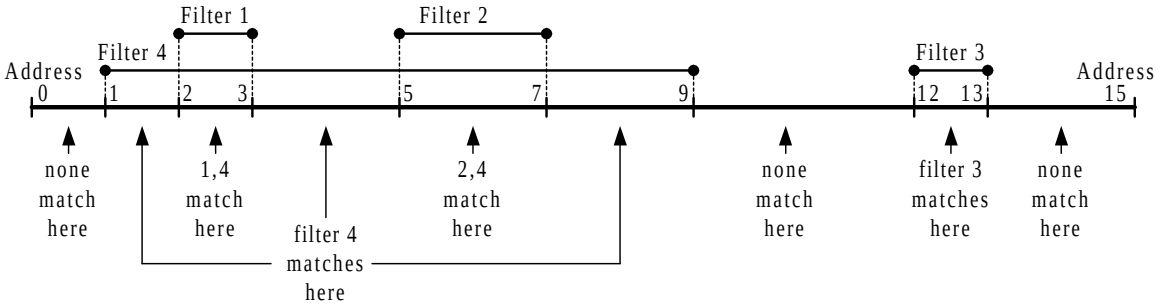

Figure 1: Filters projected onto an axis (1-D example)

same filters. In this example the addresses 1, 4, 8, and 9 all match exactly the same filters (i.e. only filter 4.) Therefore, those addresses belong in the same equivalence set. There are a total of five of these equivalence sets in this example, shown in Table 3.

Two points in the same interval always belong to the same equivalence set. Also, two intervals are in the same equivalence set if exactly the same filters project onto them.

Equivalence Class	*Filters Matched*	*Values (destination addresses) in Equivalence Set*
E_0	none	0, 10, 11, 14, 15
E_1	only 4	1, 4, 8, 9
E_2	1 and 4	2, 3
E_3	2 and 4	5..7
E_4	only 3	12, 13

Table 3: Equivalence sets for the 1-D example

To help us solve the best matching filter problem, we can precompute a table that maps each possible value for the address to the equivalence class to which it belongs. For the example we are using, this lookup table would look like Table 4.

In actual implementation, the equivalence classes are represented by integers 0, 1, 2,... instead of symbols $E_0, E_1, E2, ...$ so they can be used to index into another table. In this case, since we know the filters matching each equivalence class, we can precompute a table that maps each equivalance class to the least cost filter matched by the values in its equivalence set. The result of this is Table 5.

So, to do a lookup for a packet P with header field containing x, we would do the following: Perform a table lookup of the address x (using Table 4) to find the equivalence class to which the address x belongs. This equivalence class indicates which filters match the patcket P. Next, perform a table lookup of this equivalence class identifier (using Table 5) to determine the least-cost matching filter.

If, for example, we receive a packet with destination address 5, we first look up destination address 5 in Table 4. The fifth entry is E_3, i.e. E_3 is the equivalence class for address 5. We then look up E_3 in Table 5; the third entry is 2, indicating that Filter 2 is the least cost filter matched by addresses in E_3 (and thus by address 5.)

Of course, this 1-dimensional lookup can be streamlined, by storing the best matching filters instead of the equivalence class identifiers in Table 4. Thus, for the 1-dimensional case, the equivalence classes are not required. They are, however, a compact representation for intermediate results

Address	*Equivalence Class*
0	E_0
1	E_1
2	E_2
3	E_2
4	E_1
5	E_3
6	E_3
7	E_3
8	E_1
9	E_1
10	E_0
11	E_0
12	E_4
13	E_4
14	E_0
15	E_0

Table 4: Lookup table for the 1-D example

Equivalence Class	*Least Cost Filter Matched*
E_0	none
E_1	4
E_2	1
E_3	2
E_4	3

Table 5: Best matching filter for each equivalence class (1-D example)

(i.e. which filters have been matched so far) during classification on multiple fields; this becomes more apparent in the next example, which involves a 2-dimensional lookup.

2.2. Use of Crossproducting

Consider now an example where two header fields are used for classification. Let the fields be source and destination addresses; for simplicity, let each address be only four bits in length. The filters in this example are listed in Table 6.

Filter Number	*Destination Address*	*Source Address*	*Cost*
1	*	10*	1
2	100*	010*	2
3	10*	*	3
4	010*	010*	4

Table 6: Example 2-dimensional filter database

As before, we can project the filters onto an axis that represents the destination address, as shown at the top of Figure 2. This can be used to partition the destination address space into equivalence sets; these are indicated in the same figure and Table 7.

Similarly, we can project the filters onto an axis that represents the source address, as shown at the left of Figure 2. We use this to partition the source address space into equivalence sets, where source address values in the same set match exactly the same filters. These equivalence sets are shown in the figure and in Table 8.

A table can now be constructed mapping each destination address to the equivalence class to which it belongs; this mapping is shown in Table 9. Another table can be constructed to map each source address to the equivalence class to which it belongs; this mapping is shown in Table 10.

By looking up a packet's destination address in Table 9, we obtain an equivalence class identifier which indicates the set of filters matched by that destination address. Similarly, by looking up a

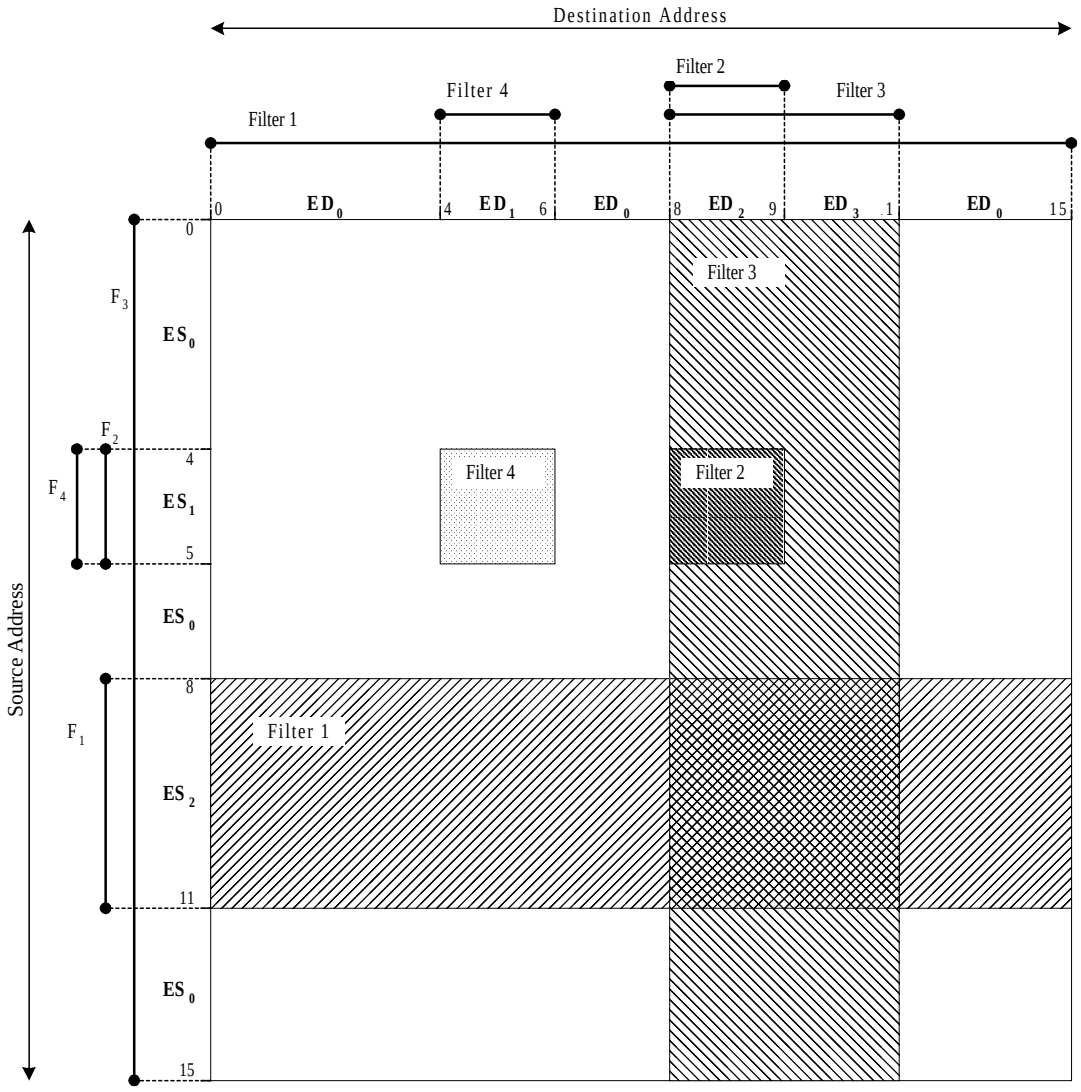

Figure 2: Filters projected onto axes (2-D example)

packet's source address in Table 10, we obtain an equivalence class identifier which indicates the set of filters matched by that source address. But, what we really want is an indication of which filters are matched by both the destination and the source addresses.

We can compute this by finding the intersection of the set of filters matched by the destination address and the set of filters matched by the source address. This, however, can be too expensive to compute at lookup time if there are many filters (if there are N filters, an N-bit wide AND operation would be needed), so we precompute the results of these intersections and store the results in a 2-dimensional table; the table is computed such that the entry $table[x][y]$ indicates the intersection of the set of filters matched in equivalence class x and the set of filters matched in equivalence class y.

Each entry in this 2-dimensional crossproducting table is used to indicate a set of matching filters. The same set of filters may occur more than one time in the table; thus it makes sense define a new set of equivalence class identifiers to represent these sets, so the table itself only contains

Equivalence Class	*Filters Matched*	*Destination Addresses in Equivalence Set*
ED_0	1 only	0..3, 6, 7, 12..15
ED_1	1 and 4	4..6
ED_2	1, 2, and 3	8, 9
ED_3	1 and 3	10, 11

Table 7: Equivalence sets for destination address (2-D example)

Equivalence Class	*Filters Matched*	*Source Addresses in Equivalence Set*
ES_0	3 only	0..3, 6, 7, 12..15
ES_1	2, 3, and 4	4, 5
ES_2	1, and 3	8..11

Table 8: Equivalence sets for source address (2-D example)

equivalence class identifiers. So, to precompute an entry $table[x][y]$ in the crossproducting table, we must do the following:

1. Look up the set of filters matched by equivalence classes x and y,
2. Compute the intersection of that set (bitwise AND),
3. Determine the equivalence class to which that result belongs; store this as $table[x][y]$.

The new equivalence classes for this example are listed in Table 11. These classes are defined during the creation of the crossproducting table, shown in Table 12, for it is only then that we know which sets of filters will occur.

Dest. Address	*Equivalence Class*
0	ED_0
1	ED_0
2	ED_0
3	ED_0
4	ED_1
5	ED_1
6	ED_0
7	ED_0
8	ED_2
9	ED_2
10	ED_3
11	ED_3
12	ED_0
13	ED_0
14	ED_0
15	ED_0

Table 9: Lookup table for destination address (2-D example)

Source Address	*Equivalence Class*
0	ES_0
1	ES_0
2	ES_0
3	ES_0
4	ES_1
5	ES_1
6	ES_0
7	ES_0
8	ES_2
9	ES_2
10	ES_2
11	ES_2
12	ES_0
13	ES_0
14	ES_0
15	ES_0

Table 10: Lookup table for source address (2-D example)

Equivalence Class	*Filters Matched*
EC_0	none
EC_1	only 2
EC_2	only 4
EC_3	2 and 3
EC_4	only 1
EC_5	1 and 2

Table 11: Equivalence classes for crossproducting table (2-D example)

	ED_0	ED_1	ED_2	ED_3
ES_0	EC_0	EC_0	EC_1	EC_1
ES_1	EC_0	EC_2	EC_3	EC_1
ES_2	EC_4	EC_4	EC_1	EC_5

Table 12: 2-dimensional crossproducting table (2-D example)

To perform classification, we need both one-dimensional lookup tables (Tables 9 and 10), the two-dimensional crossproducting table (Table 12), and the mapping from final equivalence class identifier to classifier output (Table 11.) The other tables are only needed during initialization. To see how this works, consider the following example:

Suppose a packet arrives with destination address 9 and source address 5. To classify this packet, we first look up destination address 9 in Table 9, which gives us the result ED_2. We also look up source address 5 in Table 10, giving us the result ES_1. These results, ED_2 and ES_1, indicate the filters matched by the destination address and by the source address respectively; we use these equivalence class identifiers to index into Table 12 to find which filters are matched by both destination and source addresses. In this example, we would use entry (2, 1) of Table 12, which is EC_3. Using Table 11 we can see that filters 2 and 3 were matched by the packet.

The last step (using Table 11) can be eliminated by storing the least-cost matching filter directly in the entries of Table 12; in our example, then, entry (2, 1) of that table would contain the number 2 (identifying the least cost filter matched.)

2.3. Extending to k Dimensions

Classification in two dimensions starts by finding a pair of equivalence class identifiers, and uses a precomputed 2-dimensional table to map those to a single equivalence class identifier.

In the case of three dimensions, we start by finding three equivalence class identifiers; let us call these x, y, and z. Each identifier indicates which filters are matched by the corresponding header field. To find which filters match in all three dimensions, we need to compute the intersection of these three sets of filters. Again, this intersection can be too costly to evaluate during a lookup, so we wish to precompute as much as we can.

One approach to this in RFC is to create a 3-dimensional crossproducting table, where each value $table[x][y][z]$ is precomputed by finding the intersection of the sets of filters matched in equivalence sets x, y, and z. But this approach can scale poorly in terms of memory requirements, especially when extending to more than three dimensions; thus, it is not considered in depth in this paper.

Another approach to this in RFC is to use multiple 2-dimensional crossproducting tables; this is the approach preferred in this paper. To classify packets in three dimensions, we need two such 2-dimensional crossproducting tables. The first table is computed such that $table1[x][y] = a$ where a identifies an equivalence class corresponding to the intersection of the filters matched in x and y. The second table is computed such that $table2[a][z] = b$ where b identifies an equivalence class corresponding to the intersection of the filters matched in a and z. This example is shown in Figure 3.

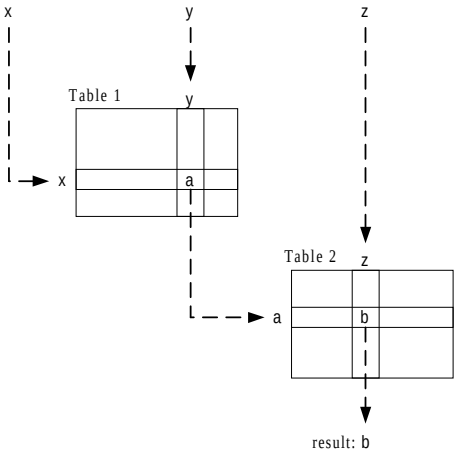

Figure 3: Reduction of three equivalence class identifiers to one

The equivalence class identified by b, then, corresponds to the set of filters matched by all three header fields. Thus, we can get the same result while generally requiring less memory than the 3-dimensional table.

This idea can be extended to handle k dimensions, by using $k-1$ separate 2-dimensional tables. Each table combines two equivalence class identifiers into one equivalence class identifier; thus, with $k-1$ two-dimensional crossproducting tables, we can go from k equivalence class identifiers (one for each field) to just one.

The order in which these identifiers are combined corresponds to a structure called a reduction tree, where each node in the tree represents a crossproducting table, and its children are the source of the equivalence class identifiers used to index into that table. Figure 4 shows an example of a reduction tree for classification using three dimensions (source address, destination address, protocol information.) A more compact representation of the same tree is shown in Figure 5; the 1-dimensional lookup tables are implied, but omitted from the figure for brevity.

Prefix matching on a large field can be performed by splitting it up into more than one chunk. This is useful for fields exceeding 16 bits in length (e.g. IP addresses), since a field W bits wide requires a table with 2^W entries to map values to equivalence classes. Fields with range matches cannot be split this way, but a method exists for transforming the range location problem into a prefix matching problem [6].

Using these techniques, we can build a classifier for the standard 5-tuple used in the Internet context. Table 13 shows one way to define the header chunks, and Figure 6 shows one of the possible reduction trees for this set of chunks.

Thus, with these extensions, what began as a simple example has been extended to become Recursive Flow Classification. Gupta and McKeown also describe an optional adjacency group optimization which can be used in some applications; that optimization can be used in conjunction with the compression scheme described in this paper, but for simplicity we omit it.

3. Reducing Storage Requirements

The primary research contribution of this paper is an exploration of ways to reduce the amount of storage needed for classification using the RFC algorithm. Here we consider a simple compression

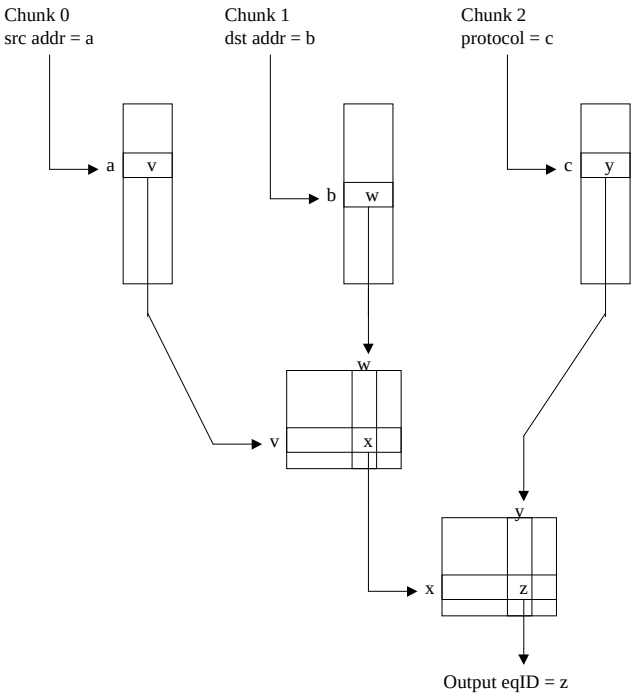

Figure 4: An example reduction tree for classifying on three fields

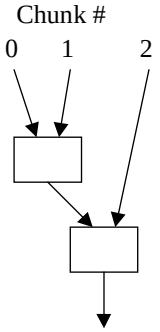

Figure 5: Simplified representation of example reduction tree

technique, an improvement on that scheme, and the effect of using different reduction trees.

To evaluate each approach, we construct a classifier using a real filter database with 159 rules. The classifier splits the header fields into 7 chunks as indicated in Table 13. We use 16-bit chunks as suggested in the RFC paper, and because the port fields require range matching, which precludes splitting them into smaller chunks. We perform classification on five header fields in this experiment, though in some papers it appears that RFC is only used for classification on four fields.

Since the choice of reduction tree affects the size of the 2-dimentional tables, and there is no immediately obvious "best choice" reduction tree, we consider all possible reduction trees for this database in each case.

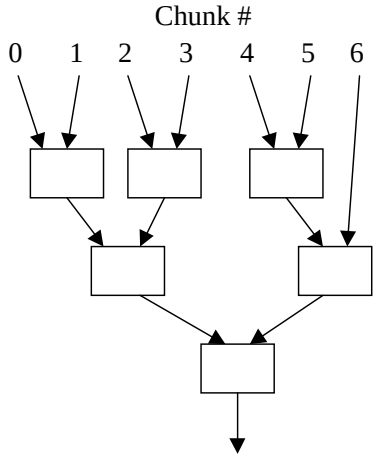

Figure 6: An example reduction tree for a typical 5-tuple classifier

Chunk Number	*Chunk Contents*
0	First 16 bits of IP source address
1	Last 16 bits of IP source address
2	First 16 bits of IP destination address
3	Last 16 bits of IP destination address
4	Source port number
5	Destination port number
6	Transport protocol number and flags

Table 13: An example set of chunks for a typical 5-tuple classifier.

3.1. Using Compressed Data Structures

The classification method just described has excellent performance with respect to lookup time, but its memory requirements can be quite high when many filters are used. Here we describe a scheme using compressed data structures to reduce the memory requirements of the lookup algorithm.

This method is based on two observations: first, that much of the storage is required for the crossproducting tables (especially for large filter databases), and secondly, that these crossproducting tables tend to have many contiguous elements repeated (again, especially true with large filter databases.) We can take advantage of this to compress the crossproducting tables, but this new representation must still allow fast lookups.

Let us consider a table stored in row-major order in an array. The original array can be represented by a compressed array and a bit vector. For each run of repeated elements, we store only one such element in the compressed array. Thus, the compressed array for $A\ A\ A\ A\ B\ B\ B\ C\ B\ B\ C\ C$ would be $A\ B\ C\ B\ C$. The bit vector has a bit corresponding to each element in the original (uncompressed) array; this bit is a 0 if that element is the first element or is the same as the previous element, and 1 otherwise; thus, the bit vector for $A\ A\ A\ A\ B\ B\ B\ C\ B\ B\ C\ C$ would be 0 0 0 0 1 0 0 1 1 0 1 0.

The bit vector is used to find the results of a lookup in the compressed array. If we want to know what the ith item was in the original array, we count the number of 1s in bit vector elements zero through i, inclusive. If there are j 1s, then we can find the result by looking at the jth element

of the compressed array. Continuing the example from the previous paragraph: To look up the 7th element of the original array $A\ A\ A\ A\ B\ B\ B\ C\ B\ B\ C\ C$, we count the 1s in bit vector elements 0 through 7; there are 2, so the answer is element 2 of the compressed array $A\ B\ C\ B\ C$, i. e. the answer is C.

Counting the number of 1s becomes expensive when the bit vector is large; to avoid performing much of this work at classification time, we use precomputation as follows: If we precompute the total of 1s set in the first W bits, the first $2W$ bits, the first $3W$ bits, etc., then at lookup time we only need to count at most $W - 1$ bits in the bit vector.

Thus, each crossproducting table can be represented efficiently by a compressed array, a bit vector with a bit for each item in the original array, and a set of precomputed counts of ones.

Experimental Results: Results are shown in Table 14. In this experiment, compression typically reduced the crossproducting table storage requirements by 39%. The overall storage requirements were typically reduced by 22%, but we expect that larger classifiers would enjoy better results. With the relatively small (159 filter) classifier in this experiment, the one-dimensional tables (which are not compressed) occupied 2,621,440 bits; in some cases, this accounted for the majority of the data structure size. But with a larger classifier, we expect that the crossproducting tables occupy a much greater fraction of the overall storage needed.

Reductions of over 80% occurred in some cases, usually involving large tables that compressed well (but, not all large tables compress well.) In a few cases, the compressed form actually required more storage than the uncompressed form; this occurs when the size reduction from the original array to the compressed array is less than the additional storage needed to hold the bit vector.

	Uncompressed size	*Compressed size*	*Overall size reduction*	*Reduction of crossproduct tables*
Largest uncompressed	89,380,018	19,926,831	77.7%	80.1%
Smallest uncompressed	3,113,672	2,914,527	6.4%	40.5%
Largest compressed	89,380,018	96,187,539	-7.6%	-7.8%
Smallest compressed	3,113,672	2,914,527	6.4%	40.5%
Most overall compression	77,233,034	15,235,167	80.3%	83.1%
Least overall compression	67,280,539	72,425,225	-7.6%	-7.9%
Average values	12,196,446	8,227,865	25.4%	36.7%
Median values	7,454,688	5,800,143	22.4%	39.5%

Table 14: Experimental compression results (table sizes in bits.) Rows 1-6 each correspond to a specific reduction tree; rows 7, 8 are average and median for these values across all reduction trees.

3.2. Improving Compression: TSP Heuristic

The compression technique just described relies on the tendency for adjacent table entries in the same row to have the same value. For this reason, some tables compress well, and others do not. For example, the table shown in Table 15 does not compress particularly well.

Since the equivalence class identifiers are assigned in an arbitrary order, it is possible to re-arrange rows and/or columns of the tables by reassigning the identifiers. Thus it is possible to re-arrange rows and columns in a table to improve compression. If we re-order the columns of Table 15 we can produce Table 16, which will result in improved compression since it has more runs of repeated elements.

	0	1	2
0	a	b	a
1	b	a	b
2	a	b	a

Table 15: Two-dimensional crossproducting table with poor compression

	0	1	2
0	a	a	b
1	b	b	a
2	a	a	b

Table 16: Two-dimensional crossproducting table with better compression

In general, there are too many ways re-arrange a large table to conduct an exhaustive search. However, there are some heuristics that usually produce good results. For example, if tables are stored in row-major order, then re-arranging rows will have little effect compared to re-arranging columns.

The question of how to re-arrange the columns for best compression can be transformed into a variation of the Traveling Salesman Problem, as follows: Let each column in the table be represented as a node in the TSP problem. The goal is to select an ordering (tour) of the columns (nodes) such that the number of elements in the compressed array (cost of the tour) is minimized.

The total cost of an ordering of columns is the number of elements in the compressed array. An element in column $i+1$ is only added to the compressed array if it differs from the element in column i of the same row; thus, the cost contribution of placing column $i+1$ immediately after column i is equal to the number of rows in which the two columns have differing entries. This way, the cost of a particular tour reflects the cost of using that ordering for columns, except for the cost of the first column itself (due to wrap-around from last column of a row to first column of the next row.)

With this definition of cost, note that cost(A,C) $\leq$ cost(A,B) + cost(B,C) (i.e. the triangle inequality applies.) Thus, TSP approximation algorithms based on a minimum spanning tree will work and can be used to find an ordering of columns that produces good results.

This generally produces better results than the naive compression scheme described earlier, but computing the TSP cost matrix can be expensive. For a 2-dimensional table with R rows and C columns, this requires O(C^2R) time and O(C^2) space.

Experimental Results: In this experiment, the TSP heuristic is used when practical given the time constraints (in this case, as long as no crossproducting table exceeded 6,400 columns), and the simpler compression technique is used otherwise. The results are shown in Table 17.

In these results, compression with the TSP heuristic typically reduced the crossproduct table storage requirements by 62%. Reductions over 85% occurred in some cases, usually involving large tables that compressed well (but, not all large tables compress well.) There are still cases where the compressed form requires slightly more storage than the uncompressed form, but not as much as when using the naive compression method.

The overall storage requirements were typically reduced by 37%. But, again, that is mainly due to the one-dimensional tables occupying a significant fraction of the data structure, due to the small size of this classifier. We expect that larger classifiers will experience more overall compression, since their crossproduct tables will occupy a larger fraction of the overall data structure size.

Furthermore, larger classifers appear to produce crossproducting tables that are more compressible. The TSP-guided compression scheme has been applied to subsets of a larger filter database, varying the number of rules each time. This time we are considering only one reduction tree, selected arbitrarily from those having average results with the other database. The results are shown in Figure 7.

	Uncompressed size	*Compressed size*	*Size reduction*	*Reduction of crossproduct tables*
Largest uncompressed	89,380,018	13,677,785	84.7%	87.3%
Smallest uncompressed	3,113,672	2,905,502	6.7%	42.3%
Largest compressed	67,963,198	73,138,696	-7.6%	-7.9%
Smallest compressed	3,113,672	2,905,502	6.7%	42.3%
Most overall compression	89,253,854	13,580,518	84.8%	87.3%
Least overall compression	73,052,072	67,868,399	-7.6%	-7.9%
Average values	12,196,446	5,976,058	38.1%	54.4%
Median values	7,454,688	4,931,011	37.7%	61.6%

Table 17: Experimental TSP-heuristic compression results (table sizes in bits.) Rows 1-6 each correspond to a specific reduction tree; rows 7, 8 are average and median for these values across all reduction trees.

These results suggest that, for typical filter database, a larger number of rules tends to result in more compressible arrays. As the number of filters increases, the reduction in size of the crossproduct table tends to increase (exceeding 60% at around 250 filters.) The sharp jumps in the overall compression efficiency in Figure 7 appear to occur when the addition of a particular rule greatly increases the size of the crossproducting tables; note that the compression ratio of the crossproducting tables remains approximately the same, but the overall compression ratio jumps since the crossproducting tables are now a larger part of the whole.

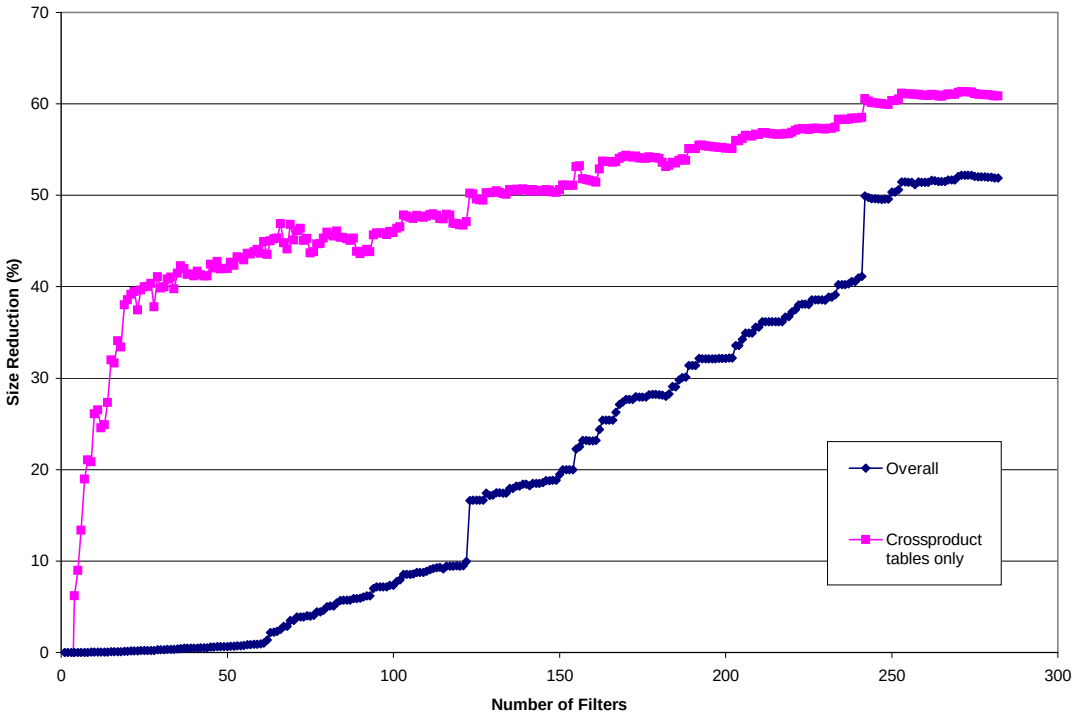

Figure 7: Compression Efficiency vs. Number of Filters.

3.3. Choice of Reduction Trees

The choice of reduction tree (explained in Section 2.3) affects the size of the data structure created. This is true for both the uncompressed representations and the compressed representations of the data structures.

The paper by Gupta and McKeown mentions two heuristics for selecting the reduction tree: (i) combining chunks with the most "correlation," e.g. the two 16-bit chunks of IP source address, as soon as possible, and (ii) combining as many chunks as possible without causing unreasonable memory consumption. For this experiment, we follow heuristic (ii) by only combining chunks two at a time. The "correlation" for heuristic (i) is not clearly defined, so we consider all possible combinations to see the results.

Experimental Results: The results collected are summarized in Table 14 and Table 17. The average uncompressed data structure required 12,196,446 bits of storage, but the most efficient one required only 3,113,672 bits; this represents a 74.5% reduction in size. So, although compression can help, proper selection of the reduction tree can help more.

4. Conclusion

As noted in the Gupta and McKeown paper, it is easy to perform packet classification at high speed using large amounts of storage, or at low speed using small amounts of storage. The Recursive Flow Classification algorithm they describe exploits structure and redundancy found in typical filter databases; this allows fast packet classification with reasonable storage requirements for databases with thousands of rules.

The simple compression technique described in Section 3.1 reduced storage requirements for the classifier's crossproduct tables by 37% on average in the experiment. The overall storage requirements were reduced by 25% on average, but it appears that larger classifiers will be more compressible.

The efficiency of the compression can be improved by the TSP heuristic described in Section 3.2. This reduced storage requirements by 54% on average in the experiment. The efficiency of compression appears to increase with larger filter databases, but for sufficiently large filter databases (where compression is really needed) the computational cost of this heuristic becomes excessive.

The most benefit, in terms of reducing storage requirements, comes from proper selection of the reduction tree. Optimal reduction tree selection reduced the storage requirements by 74% over the average size in our experiment. How to select a good reduction tree is left as an open question for future research.

Acknowledgements: The author would like to thank George Varghese for the direction of looking at compressing the RFC tables, Subhash Suri for feedback on the TSP heuristic, and Jonathan Turner for help writing the paper.

References

[1] F. Baboescu and G. Varghese, "Scalable packet classification," Proc. of ACM Sigcomm '01, September 2001.

[2] F. Baboescu, S. Singh, and G. Varghese, "Packet Classification for Core Routers: Is there an alternative to CAMs?" UCSD Tech report.

[3] A. Brodnik, S. Carlsson, M. Degemark, S. Pink. "Small Forwarding Tables for Fast Routing Lookups," Proc. ACM SIGCOMM 1997, pp. 3-14, Cannes, France.

[4] M. M. Buddhikot, S. Suri, and M. Waldvogel, "Space decomposition techniques for fast layer-4 switching," Proc. of PHSN, August 1999.

[5] W. Eatherton, "Hardware-based internet protocol prefix lookups," MS Thesis, Washington University in St. Louis, Electrical Engineering Department, May 1999.

[6] A. Feldman and S. Muthukrishnan, "Tradeoffs for packet classification," Proc. of Infocomm, March 2000.

[7] P. Gupta and N. McKeown "Packet Classification on Multiple Fields," in *Proc. ACM Sigcomm '99,* Sept. 1999.

[8] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," Proceedings of the Conference on Computer Communications (IEEE INFOCOMM), (San Francisco, California), vol. 3, pp. 1241-1248, March/April 1998.

[9] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," Proc. of Hot Interconnects VII, August 1999.

[10] T. V. Lakshman and D. Stidialis, "High speed policy-based packet forwarding using efficient multi-dimensional range matching," Proc. of ACM Sigcomm '98, September 1998.

[11] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," Proceedings of the Conference on Computer Communications (IEEE INFOCOMM), (San Francisco, California), vol. 3, pp. 1248-1256, March/April 1998.

[12] C. Matsumoto, "Cam vendors consider algorithmic alternatives," EE Times, May 2002.

[13] V. Srinivasan and G. Varghese, "Fast IP Lookups using Controlled Prefix Expansion," Proc. ACM Sigmetrics, June 1998.

[14] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer 4 switching," Proc. of ACM Sigcomm '98, September 1998.

[15] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," Proc. of ACM Sigcomm '99, September 1999.

[16] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, "Scalable High-Speed IP Routing Lookups," Proc. ACM SIGCOMM 1998, pp.25-36, Cannes, France.

[17] Packet Classification Repository. http://www.cs.ucsd.edu/~baboescu/repository/