Packet Classification via Improved Space Decomposition Techniques

Filippo Geraci, Marco Pellegrini, Paolo Pisati
IIT-CNR, Pisa, {filippo.geraci,marco.pellegrini,paolo.pisati}@iit.cnr.it
Luigi Rizzo
Dip.Ing.Informazione, Univ. di Pisa, rizzo@iet.unipi.it

***Abstract*— Packet Classification is a common task in modern Internet routers. The goal is to classify packets into "classes" or "flows" according to some ruleset that looks at multiple fields of each packet. Differentiated actions can then be applied to the traffic depending on the result of the classification.**

Even though rulesets can be expressed in a relatively compact way by using high level languages, the resulting decision trees can partition the search space (the set of possible attribute values) in a potentially very large (10^6 and more) number of regions. This calls for methods that scale to such large problem sizes, though the only scalable proposal in the literature so far is the one based on a Fat Inverted Segment Tree [1].

In this paper we propose a new geometric technique called *G-filter* for packet classification on d dimensions. G-filter is based on an improved space decomposition technique. In addition to a theoretical analysis showing that classification in G-filter has $O(1)$ time complexity and slightly super-linear space in the number of rules, we provide thorough experiments showing that the constants involved are extremely small on a wide range of problem sizes, and that G-filter improve the best results in the literature for large problem sizes, and is competitive for small sizes as well.

***Index Terms*— System design**

I. Introduction

The problem of packet classification has received much attention in recent years, due to its widespread application to different types of network equipment. In a nutshell, the problem is to classify packets into "classes" or "flows" (depending on the granularity) by looking at one or more packet attributes. This is normally done by routers (doing a next-hop lookup), firewalls (filtering traffic), shapers and policers (to enforce traffic limitations), NAT boxes, and queue management systems.

The classification is done according to a *ruleset*, which can be specified in different languages[2], [3], [4], [5], [6], as shown in Section II-A. Because classification is done for many different purposes, and on different sets of packet attributes, it is unclear that any single approach can suit all purposes. Sec. II-B, shows some of the solutions proposed in the literature, with different areas of applicability.

One possible approach is to map the problem into a geometric point location problem in a multi-dimensional space. The space is partitioned into a number of possibly overlapping regions, each associated with an integer indicating its priority. The number of regions can become very large, up to 10^6 and more, resulting from the number of possible paths in the decision tree generated by the specification of the ruleset. In this formulation, the problem then becomes finding the region with highest priority to which a point belongs. Theoretical results by [7] show how to do classification through point location for a 2-D space in $O(1)$ time using slightly super-linear storage. These results have been extended in [8] to handle d-dimensional rules, for any arbitrary, but constant, value of d. But probably more important than the asymptotic complexity, in a practical implementation, the constants hidden in the $O()$ notation become of fundamental importance.

The contribution of this paper is a novel geometric algorithm, called G-filter, for multidimensional packet classification. By theoretical analysis we show that G-filter has $O(1)$ classification time and slightly superlinear space in the number of rules. More interestingly from a practical point of view, through extensive simulations on datasets with different properties, we show that G-filter outperforms the best published results in the literature [1] on large datasets, and remains competitive also for small datasets.

The paper is structured as follows. In Sec. II we formalize the problem of packet classification. In Sec. II-A we briefly discuss filter specification languages. Sec. II-B presents the most relevant related work. Section III presents the G-filter algorithm, followed in Sec. III-C

Work partially supported by the Italian Registry of ccTLD .it.

by a theoretical analysis of its worst case performance. Sec. IV shows, through simulation, that G-filter is practical and improves other proposals in the literature.

II. Problem definition and related work

We can state the packet classification problem as follows: given a packet q (the "query point" in our representation of the problem) made of a set of attributes $q_1, ..q_d$ (each q_i mapped to an integer in the range $U = [0 \dots 2^w - 1]$), and a set H of rules specifying a partition of the attribute space U^d into different regions (classes), we want to associate the packet to a *class* depending on the value of its attributes. Typical attributes can be source and destination addresses, protocol type, port numbers (together, these attributes are called the "5-tuple"), protocol flags, and possibly other attributes such as packet size and even meta-attributes (e.g. source or destination interface, etc.).

The classification result is typically associated to the action to be performed on the packet. For a firewall, it could be as simple as accept or deny a packet; for a more complex system, the classification result might be used to aggregate the packet into logical *flows* (to be passed to separate queues, or be subject to shaping or policing) or simply to collect statistics.

A. Ruleset specification

The ruleset that partitions the attribute space into classes can be specified in different ways. A common approach is to use a sequential list of n *rules* of the form $< class, r_1, r_2, \dots, r_d >$ where $r_1, \dots, r_d$ are ranges specifying a *hyper-rectangular* region in the attribute space, and $class$ is the result of the classification. The classifier will scan the list, in textual order, against incoming packets stopping the search at the first rule whose region contains the packet's attributes. This is the approach used by Cisco's ACLs [5], and in the basic format of Juniper [6] or ipfw [2] rules. Basic ipfilter [3] rules are similar, but there the search always continues to the end[1] and the classifier returns the last matching rule.

The fixed rule search ordering is equivalent to associating a *priority* field to each rule; this formulation of the ruleset makes it possible to approach the problem with more efficient algorithms than the linear scan of the ruleset, which has $O(n)$ time complexity.

[1]unless the rule contains a "quick" keyword to terminate the search early.

In practice, however, ruleset specification languages tend to be a lot more complex than the simple list of rules described above.

First, we could have negations on the ranges of some or all the attributes (e.g. `src-port 0-1023 not dst-port 0-1023`). Some techniques can easily deal with negations, other may not, or will suffer a severe space overhead.

Second, some classifiers (e.g. those used in stateful firewalls) can generate or remove rules dynamically. Fortunately these tend to have a uniform format (e.g. because they are generated from a specific template) and so they can be dealt with separately from the static part of the ruleset.

Finally, the independent rules described so far tend to be very redundant – e.g. many rules will use the same protocol and port ranges, and differentiate on other attributes.

If rulesets are generated manually (as it is often the case), it is extremely convenient to use a structured ruleset specification language, which allows partial evaluation of the attributes to be performed. This is supported e.g. by Juniper [6] or ipfw [2] rules, where after a match the classification may continue by jumping to a different point in the ruleset (e.g. in ipfw syntax, `skipto 1000 proto tcp src-port 80`).

It is still possible to transform a structured ruleset into a flat one (where rules can be evaluated independently), but at the price of a (possibly large) increase in the ruleset size. On the other hand, this transformation can be worthwhile as it can open the way to the use of more efficient classification algorithms. So this calls for packet classification algorithms that can work efficiently on very large rulesets.

B. Related work

The packet classification problem has been extensively studied recently. The naive approach to packet classification is to scan sequentially the rule list until a match is found. The scalability of this solution is generally poor, as the search time is proportional to the length of the longest path in the rule list.

The main solutions to improve the search times use various combinations of one or more of the following: (a) hardware-based solutions [9], (b) specialized data structures [10], (c) geometry-based algorithms [7], and (d) heuristics [11].

Hardware-based solutions using CAMs can be used to exploit the parallelism in the hardware to look up multiple rules in parallel. They are limited to small rulesets because of cost, power and size limitations of CAMs. Other hardware based solutions are described in [12], but still limited to a small number of rules.

If the rulesets language allows jumps, one can structure the ruleset as a trie, with a classification time $O(B)$ where B is the total number of bits on all dimensions. This value can still be exceedingly large (e.g. for the 5-tuple in IPv4, $B = 104$, and this motivates the research on algorithms that have lower complexity with typical rulesets.

Aggregated Bit Vector(ABV)[13] solves the problem with d independent lookups on one dimension, followed by a combining phase. For each dimension, a lookup is done using a trie, and returning a list of all matching rules on that dimensions. The final result is then computed by finding the rule with highest priority which is present in all lists. Because the amount of memory consumed for storing the lists can be extremely large, ABV devotes a lot of effort in reducing the memory overhead, by representing the list using a compressed bit vector.
Unfortunately, just navigating the tries still requires $O(B)$ time, and the compression of the rule lists is not as effective as one would like.

A geometry-based algorithm was proposed by Feldmann et al. [1], introducing a data structure called FIS Tree (Fat Inverted Segment Tree). Here, the problem is approached one dimension at a time. FIS partitions the first dimension with the endpoints of the projection of the rules on that dimension. Each of the segments is then partitioned, according to the remaining dimensions of the rules covering each segment, into a number of a $(d-1)-$dimensional regions. These can be looked up using a $(d-1)-$dimensional version of the algorithm. To avoid an $O(N^2)$ explosion of the storage requirements, the $d-1$ dimensional regions are linked in a Fat Inverted Segment Tree (*FIS tree*, which gives the name to the algorithm) of bounded depth, and the common partitions of the regions are pushed up in the FIS tree. So, the $(d-1)-$dimensional lookup is repeated (but only a bounded number of times) on each of the nodes of the FIS tree from the leaf to the root.
To date, FIS tree is the algorithm that scales best with the number of rules.

Gupta and McKeown[14] proposed a heuristic approach called RFC (Recursive Flow Classification). The main idea is that packet classification involves mapping S bits in the packet header to $T \ll S$ bits of action identifier (this is done via a lookup table). These partial identifiers are then combined, and the reduction process continues until the final result is reached. The depth of the structure is an input parameter of the algorithm, and influences the classification time. An advantage of RFC is that the various lookup stages can be pipelined, so in a hardware implementation, the classifier can have a very high throughput. Scalability to medium or large rulesets is still an issue though.

III. G-Filter

Our proposal falls in the category of geometry-based solutions, and it is based on a novel recursive partitioning of the search space which has constant depth and modest space overhead.

Let $U = [0 \dots 2^w - 1]$ be the set of possible values of the packet's coordinates, and U^d a $d-$dimensional space U^d called the *universe* and representing all possible values of the packets' attributes. Given a set H of n rules, in our algorithm we map rules $h \in H$ to hyper-rectangular regions $R(h) =< R_1(h), \dots, R_d(h) >\in U^d$, regions x of the search space to hypercubes $I(x) =< I_1(x), \dots, I_d(x) >\in U^d$, and packets to be classified to points $< q_1, \dots, q_d >\in U^d$. The result of the classification is the rule with the highest priority among those containing the query point.

The algorithm is made of two parts: construction of the search data structure for a given region of the search space, and the actual packet classification. In the latter, once we have determined that a packet belongs to a given region (initially the entire universe), we use the data structure associated to that region to perform the classification.

A. Construction of the data structure

The input for the algorithm that constructs the search data structure is a region x of the search space, and a list $H(x)$ of rules potentially interesting the region x. The output is a pointer to a data structure $D^{(d)}(x, H(x))$ constructed by the algorithm. Initially, the algorithm starts with the entire ruleset ($H(root) = H$) on the entire universe ($I(root) = U^d$).

The first step of the algorithm is to partition rules $h \in H(x)$ in the following sets, with each rule belonging to only one set:

1) if h does not intersect x, it is discarded (a query point in region x will never match the rule);
2) otherwise, if h covers the entire region x, it becomes part of the set $cover(x)$ of *cover* rules;
3) otherwise, if the projection $R_j(h)$ of h on axis j entirely covers the projection $I_j(x)$ of the region x on the same axis, h becomes part of the set $FB_j(x)$ of *fallback* rules on axis j (if h satisfies

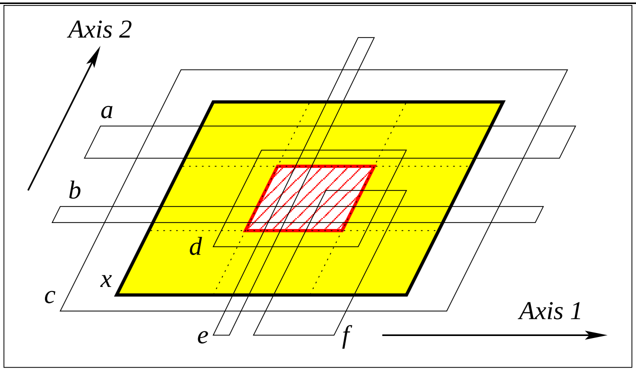

Fig. 1. An example of the construction process in a 2-d space. For the main region x, $c \in cover(x)$, $a, b \in FB_1(x)$, $e \in FB_2(x)$, $d, f \in cross(x)$. Of these, for the central subregion y, $d \in cover(y)$, $f \in cross(y)$.

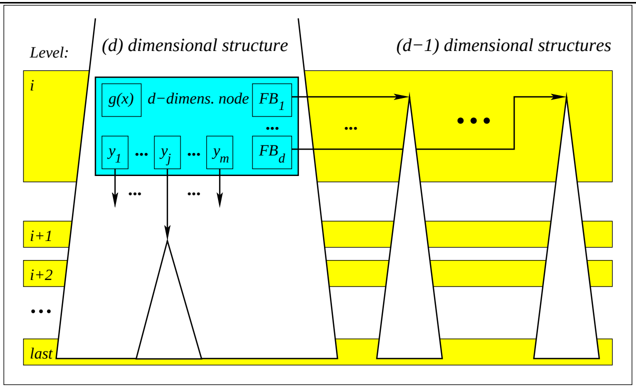

Fig. 2. The content of each node and its references to other nodes and fallback data structures.

this property for more than one axis, we arbitrarily pick one);

4) otherwise, rule h becomes part of the set $cross(x)$ of *cross* rules, which intersect x (i.e. have at least one vertex in x) but do not fall in any of the other categories. the set $cross(x)$ of *cross* rules.

The partition reflects the relation of rules with query points q belonging to region x. Fig. 1 shows a 2-d example of the relation between rules and regions.

Cover rules have the property that any packet $q \in x$ matches all rules in $cover(x)$. The only information we need to remember from this set is the rule $g(x)$ with the highest priority in $cover(x)$, as this will be a potential result for the classification.

For fallback rules, we know that if $q \in x$, then the $j-$th coordinate of q is within the range $R_j(h)$ of all the rules in the set $FB_j(x)$. So q will match a rule $h \in FB_j(x)$ if and only if its remaining $d-1$ coordinates are contained in the remaining $d-1$ ranges of the rule. This is equivalent to finding whether the projection of q *along*[2] axis j, $P_j(q)$ (which is contained in the projection $P_j(x)$) matches the projection $P_j(h)$ of the rule along axis j. So the problem reduces to a classification problem in a $(d-1)-$dimensional region.

Finally, for cross rules, the fact that $q \in x$ does not tell us anything about its possible matching with cross rules. So we need to refine the search, and we do that by by partitioning region x into m regions of uniform size and shape, and recursively constructing the structures $D^{(d)}(y_i, cross(x))$.

With this in mind, if after the rule partitioning the region has no cross, cover or fallback rules, then the construction is complete and the algorithm returns a NULL pointer. Otherwise the algorithm creates (and returns a reference to) a root *node* of the data structure $D^{(d)}(x, H(x))$ with the following information:

- a reference to rule $g(x)$, the rule with highest priority in $cover(x)$;
- d references to the $(d-1)-$dimensional structures $D^{(d-1)}(x, FB_j(x))$, recursively constructed for the fallback regions;
- m references to the (recursively constructed) structures $D^{(d)}(y_i, cross(x))$.

The construction terminates when a region has size 1, because any rules intersecting such a region must be a cross rule. As an optimization, if the total number of fallback and cross rules is smaller than some threshold t, we can avoid the recursive construction and instead store the highest covering rule and the fallback/cross rules into an array. Storagewise, this is effective if $t < m$. In terms of classification times, t should be reasonably small.

Note that G-filter is not restricted to hyper-rectangular rules. We can use rules representing arbitrarily shaped regions, even non connected ones, as long as the rule classification procedure is able to correctly process them. This is extremely useful in practice, as it is often the case, in a ruleset, that rules have negations on individual dimensions or possibly even on the entire region (e.g. `not (src-ip 10.0.0/8 and not dst-ip 10.0.0.0/8`).

Fig. 2 gives a pictorial representation of the search data structure, showing the content of each node and its references to the fallback data structure and to nodes at the next level. We can think of the entire data structure as a main tree with one node per region constituting the

[2]Note that a projection *along* axis j of a $d-$dimensional region produces a $(d-1)-$dimensional region with all coordinates but the one on axis j. This is different from the projection *on* axis j that we have used to determine if a rule belongs to the fallback set – in the latter, the projection produces a $1-$dimensional range which corresponds to the coordinates of the object on axis j.

$d-$dimensional data structure, and references to $(d-1)-$dimensional fallback structures from each node.

B. *Classification*

As a result of the previous construction, the classification can be performed as a recursive process on the data structure $D^{(d)}(root, H)$. At each node (initially the root), we perform d recursive queries on the $(d-1)-$dimensional fallback structures, one recursive query on the region $y_i | q \in y_i$, and return the highest priority rule among $g(x)$ and the rules returned by the $d+1$ recursive queries. In practice, the recursive query on region y_i can be easily transformed into an iterative one with trivial tail-recursion elimination techniques, so it is convenient to think of the classification process as a walk on the $d-$dimensional tree, visiting one node per level.

C. *Theoretical analysis*

In this Section we investigate the asymptotic time and space complexity of our algorithm. To simplify the analysis, we have used a single parameter k to control the splitting of the region in the recursive construction, so all regions are always partitioned into $m = k^d$ hypercubes. In an actual implementation of the algorithm, however, one would change m depending on the number of rules, the number of dimensions, and the size of the regions, to achieve the best space/time tradeoff. In the experimental Section we have studied these tradeoffs.

We recall that we cast the problem in a general geometric setting, and the problem we analyze is the following:

> Given an input set H of n hyper-rectangles in U^d, build a data structure $D^{(d)}(U^d, H)$ to compute efficiently $\text{argmax}_{h \in H_q} priority(h)$ (max priority query) where $H_q = \{h \in H | q \in h\}$.

1) Main result: The main theoretical result is the following Theorem:

Theorem 1: For an integer w, let $U = [0, .., 2^w - 1]$ be the set of binary numbers of w bits. Let H be a set of n hyper-rectangles in U^d and k a parameter, $1 \leq k \leq n$. We can build a data structure $D^{(d)}(U^d, H)$ using storage $O(nk^{f(d)} \log_k^d |U|)$, answering max priority queries in time $O(\log_k^d |U|)$. The constants hidden in the big-Oh notation depend on d.

Remarks:

1) The parameters of the analysis are only the attribute size, w, the decomposition parameter, k, and the number of rules, n. d is considered a constant, although an

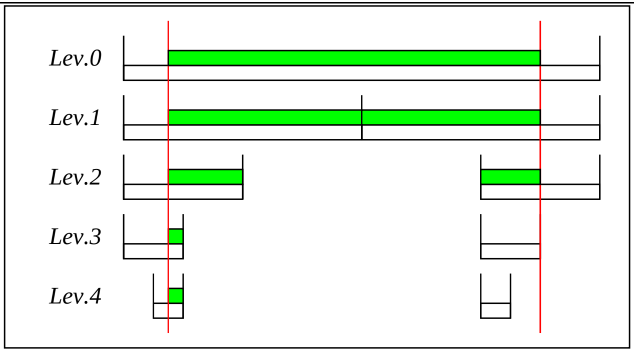

Fig. 3. A one-dimensional example of the relation between cross rules and vertices of the rules in H. The white rectangles represent the regions, at each level, for which a certain rule (the shaded rectangle) ends up in the cross set. There are at most 2 such regions per level (or 2^d in the $d-$dimensional case), and fewer when one of the vertices coincides with a boundary between two regions.

arbitrary one.

2) The function $f(d)$, which will be specified later, grows roughly as $d^2/2$.

Proof: The proof is by induction on the dimensions. The algorithm to build $D^{(d)}(U^d, H)$ is the one described in Sec. III, with no arrays (they do not change the asymptotic time-space complexity of the algorithm), and a uniform partitioning of the search region in $m = k^d$ cells at each level of algorithm. Hence, from the description at the end of Sec. III-A, and remembering that in this analysis d is considered a constant, the storage required at each node of the data structure is $O(k^d)$.

Because of the uniform decomposition of the tree, the recursion depth (*levels*) of the classification algorithm on the $d-$dimensional structure is $\log_{k^d}(|U|^d) = \log_k |U|$.

The next region y_i to visit in the $d-$dimensional data structure can be determined in $O(d)$ (i.e. constant) time by computing the indexes $(q_j - start(I_j(x)))$ div k of the $d-$dimensional array where the pointer to the data structures are stored.

Finally, we will use the following properties of cross rules (see Fig. 3): i) at any level, a vertex of a [hyper-rectangle associated to a] cross rule must correspond to a vertex of a rule in H, and ii) a vertex of a rule in H can correspond to at most one vertex of a cross rule at each level. Remembering that a hyper-rectangle in $d-$dimensions has 2^d vertices, we can have at most $n2^d$ *active* regions (i.e. those for which a recursive decomposition is required) at each level in the $d-$dimensional data structures.

We are now ready to complete the proof for the case $d = 1$, $d = 2$ and then for the generic case.

One-dimensional case: here there are no fallback sets, so the data structure is a $k-$ary tree with at most $\log_k |U|$ levels.
Query time. At each level of the tree the algorithm takes constant time, so the query time is $O(\log_k |U|)$.
Storage. The main tree has at most $2n$ active nodes per level, each requiring $O(k)$ storage. The total storage then becomes $O(nk \log_k |U|)$.

This is essentially the starting static $1-$dimensional result in [7] which we restate in a different language so to make clearer the line of reasoning leading to the multi-dimensional extensions.

Bi-dimensional case: (this case is only discussed for ease of visualization, as it is already covered in the general case) here there are two fallback sets per region.
Query time. At each level of the tree the algorithm takes constant time to locate y_i, plus it must compute two $O(\log_k |U|)$ queries on the fallback data structures. The total query time over the $\log_k |U|$ levels then becomes $O(\log_k^2 |U|)$.
Storage. The main tree has at most $4n$ active nodes at each level, each requiring $O(k^2)$ storage. So the main tree, without the fallback data structures, requires $O(nk^2 \log_k |U|)$ storage.

Consider now all active nodes at level $l-1$, and let us estimate the total size of a fallback set on one dimension at level l. A single input hyper-rectangle h at $x = parent(y)$ contributes to at most $2k$ fallback sets on one dimension among all nodes that are children of x. Moreover, at level $l-1$, h appears only 4 times since vertices are partitioned. Denoting with m_i the cardinality of the fallback set of node i at level l, we have that for every i, $m_i < n$, moreover summing on level l, $\sum_i m_i \leq 8kn$. All auxiliary data structures at level l cost order of:

$$\sum_i m_i k \log_k |U| \leq 8k^2 n \log_k |U|.$$

Summing over all levels we have that the total size of all auxiliary data structures is $O(nk^2 \log_k^2 |U|)$. So the overall storage is $O(nk^2 \log_k^2 |U|)$.

General case: the argument is inductive on the dimension. We assume that the $(d-1)-$dimensional structure uses storage $O(nk^{f(d-1)} \log_k^{d-1} |U|)$ to answer queries in time $O(\log_k^{d-1} |U|)$, and we use that to prove the same bounds for the d-dimensional structure.
Query time. At each level of the tree the algorithm takes $O(d)$ (i.e. constant) time, plus d queries on the $(d-1)-$dimensional fallback structures, each requiring $O(\log_k^{d-1} |U|)$ time. The total query time is then

$$O(\log_k |U|) \times d \times O(\log_k^{d-1} |U|) = O(\log_k^d |U|)$$

(once again remember that d is considered a constant thus it disappears in the $O()$ notation as a multiplying factor).
Storage. We have at most $2^d n$ active nodes at each level, each requiring $O(k^d)$ storage, for a total size of the main tree of $O(nk^d \log_k |U|)$.

Consider now all active nodes at level $l-1$, and let us estimate the the total size of the input sets at level l. A single input hyper-rectangle h at $x = parent(y)$ contributes to at most $2dk^{d-1}$ sets among all nodes that are children of x. Moreover, at level $l-1$, h appears only 2^d times since vertices are partitioned. Denoting with m_i the cardinality of the input sets of node i at level l we have that, for every i, $m_i < n$. Summing on level l: $\sum_i m_i \leq 2^{d+1} dk^{d-1} n$. All auxiliary data structures at level l cost order of:

$$\sum_i m_i k^{f(d-1)} \log_k^{d-1} |U| \leq$$
$$\leq 2^{d+1} dk^{d-1} nk^{f(d-1)} \log_k^{d-1} |U|$$

Now defining f recursively as $f(1) = 1$, $f(d) = f(d-1) + d - 1$ we have a bound: $O(nk^{f(d)} \log_k^{d-1} |U|)$. Summing over all levels we have that the total size of all auxiliary data structures is: $O(nk^{f(d)} \log_k^d |U|)$. ∎

2) How to reach constant query time: Now, choosing $k = n^{\epsilon/f(d)}$, for a small value $\epsilon > 0$ and using the additional assumption $n > |U|^{1/C}$, which is justified in practice, we have the following corollary:

Corollary 1: For an integer w, let $U = [0, .., 2^w - 1]$ be the set of binary numbers of w bits. Let H be a set of n hyper-rectangles in U^d, and $n \geq |U|^{1/C}$. We can build a data structure $D(H)$ using storage $O(n^{1+\epsilon})$ answering max priority queries in time $O(1)$.
The constants hidden in the big-Oh notation depend on d, ϵ and C, but not on w and n.

IV. Experimental results and comparison with other schemes

The theoretical analysis of the previous section only tells us that we can achieve constant query time with slightly superlinear storage.

The purpose of this section is to investigate, through simulation, what are the constants involved in the $O()$ notation for both query and storage, for some representative rulesets, and to compare the performance of our scheme with other significant proposals in the literature.

A. Selected algorithms

For our tests, we have compared G-filter with 3 other algorithms, which are thought to be representative of the state of the art, and already illustrated in Sec. II-B:

ABV is the algorithm proposed in [13]. We used the code from the authors of the algorithm to run the experiments on our rulesets.

RFC is a heuristic approach proposed in [15]. Once again, we used the code supplied by the authors of the algorithm to run the experiments on our rulesets.

FIS tree is a geometric approach proposed in [1]. Because neither the code nor the rulesets used for the experiments were made available by the authors, we have implemented the algorithm ourselves, and validated our implementation against the published results using synthetic rulesets (see Sec. IV-C) with the same features. Table I compares the memory and time performance of our implementation with the one in [1] on rulesets of the same size. The results are reasonably close. Therefore we consider our code as a valid implementation of the FIS tree algorithm.

Note that while G-Filter has good scalability properties with the number of dimensions and ruleset sizes, this is not the case for some of the other algorithms we compare it to. As a consequence, in this paper we limited our experiments to the 2−dimensional case. Furthermore, our focus was on storage and time used at query time, so we did not investigate the cost of the rule preprocessing phase to compute the data structures used at query time.

Finally, some of the algorithms have some tunable parameters resulting in different storage-time tradeoffs. When this was the case, we have tried a number of different values, but we omit in our graphs and tables the *dominated* points, i.e. those for which both space and time are worse than for some other experiment.

B. Metrics

The two main metrics we computed are the storage used by the data structures, and the *worst case* classification time. Storage is simply expressed as the occupation, in bytes, of the data structures used by the classification algorithm.

The time metric requires a more detailed discussion. In all the algorithms we compare the classification reduces to a navigation on a linked data structure or searches in a hash table. So the classification time is essentially dominated by the number and type of memory accesses. As a consequence, rather than measuring times, we express the classification performance in terms of the *worst case* number of memory accesses.

Rules	2 levels		3 levels		
×1000	m_f	t_m	I_{elem}	m_f	t_m
34	4.9	12	0.16	4.8	11
	2.8	13	0.11	2.7	14
67	4.5	11	0.31	4.1	12
	4.2	14	0.31	3.8	15
78	4.4	13	0.36	3.9	15
	4.1	13	0.30	3.1	16
135	4.2	12	0.61	3.8	14
	6.9	16	0.72	4.3	17
149	4.3	12	0.69	3.3	15
	4.7	14	0.64	3.5	16
150	4.1	14	0.69	3.3	15
	4.2	16	0.67	3.7	17
200	3.5	14	0.90	3.0	17
	2.9	15	0.62	2.6	18
212	4.0	16	0.92	3.1	15
	4.4	16	0.92	3.6	21
460	4.5	13	1.97	3.1	17
	4.1	15	1.84	3.5	18
540	3.0	16	2.26	2.6	19
	3.6	16	1.86	2.9	20
1090	2.3	18	4.36	1.9	21
	4.0	16	4.36	3.2	21
1150	2.6	17	4.54	5.0	16
	5.2	16	4.94	3.6	18
1180	6.2	14	4.72	4.8	17
	6.2	16	5.86	4.1	18
1310	5.4	15	5.18	1.8	22
	4.6	17	5.70	3.6	22

TABLE I
COMPARISON FOR LARGE DATA-SET. ABOVE: OUR FIS CODE, BELOW: ORIGINAL FIS DATA. m_f IS THE MEMORY FACTOR, t_m IS THE NUMBER OF ACCESSES IN THE WORST CASE, I_{elem} IS THE NUMBER OF ELEMENTARY INTERVALS.

Especially for large data structures, or for software based implementations, one can reasonably assume that if the algorithm accesses a small number of adjacent memory location, the access time is dominated by the latency of the first access (e.g. to start a burst transfer from a DRAM, or fill a cache line) and the remaining accesses (within the size of a cache line) come at almost no cost. This assumption is made by several authors (e.g. [1]) in evaluating the performance of their schemes.

Then, to make a fair comparison of the results, we count the number of accesses in two ways: one is the number of 32-bit words accessed by the classification algorithm, the other is the number of "cache line" accesses, where we count multiple accesses to the same 32-byte cache line as a single memory access. Although there are more characteristics of the access pattern that influence performance (e.g. whether accesses can be pipelined or parallelised, etc.), these two numbers give reasonable bounds for the performance of the various algorithms.

1) Determining the worst-case number of accesses: Counting the worst-case number of memory access is relatively simple in RFC (where it is a structural parameter set at build time), and ABV (where it corresponds to the longest paths in the tries, and can be derived via static analysis).

The task is slightly harder for FIS Tree, and especially for G-filter where at each level we need to perform recursive queries on the fallback data structures. Just summing the max number of accesses at all levels and for all fallback structures would yield too pessimistic results, as it would not take into account the correlations between the search paths of a single query. Thus we resort to a more refined methodology, which consists in identifying, for each algorithm, a set S of ”representative queries” for a given data structure, with the property that all combinatorially different queries are represented in S. Determining the worst case number of accesses requires: executing those queries, measuring the number of memory accesses, and returning the largest value.

In the 2-dimensional version of the G-filter data structure, we have a collection of 2D grids (the search space partitioning) and a collection of input rectangles. We compute all intersection points of all the grids with the boundaries of all the rectangles, all vertices of the grids, and all intersections of the boundaries of two input rectangles. This constitutes the representative set of queries S for G-filter. To prove it we use a continuity argument: consider a generic 2D query point q and move it without crossing any grid line or any rule boundary until it touches two lines. During such move the combinatorial path of the nodes of the data structure visited for solving the query do not change and the final position of the query is one of the points in S. Note that S depends both on the ruleset and on the specific data structure.

For the FIS tree the set of representative queries S is given by simply extending the sides of all rectangular rules into full lines and taking the intersections of pairs of such lines.

C. Rulesets

We have conducted our experiments with two types of rulesets: small rulesets and large rulesets.

Small rulesets are derived from actual firewall rulesets deployed by organizations of moderate size. They are typically constructed by hand, with an original size of 50-100 of rules (which expand to a few hundreds in the goto-less rule format supported by the classifiers in the literature). These rulesets include a large number of rules with wildcards on one dimension, which are commonly used to allow or deny all access to specific machines or subnets, irrespective of the other endpoint of the communication.

Large rulesets are instead meant to be representative of the classifiers installed in large ISP routers, and the goal is to evaluate the performance of the algorithm when dealing with up to a million rules. Clearly, such large rulesets cannot be constructed by hand, so we synthesized them using a technique similar to the one used in [1], which is meant to resemble the structure of a ruleset used for flow classification. This approach was also necessary to validate our implementation against the published results for the FIS tree, for which neither the code nor the experimental rulesets were available.

The approach used to generate a large (up to 10^6 and more rules) ruleset is to create rules with source and destination ranges corresponding to prefixes taken from a large routing table (in our case a 74k snapshot of MAE West). In addition to this table, the ruleset generator takes as input the desired ruleset size, and a histogram of the source and destination prefix length distribution, similar to the one shown in Fig. 4 (which in turn resembles the one used in [1]). As a result of this process, we have generated rulesets that range from a few thousands to over a million rules used in our experiments.

D. Parameter tuning

The setting of the various tunable parameters in the experiments is the following.

For RFC we set the number of hash table accesses to 7, and the maximum size of the hash table to 20M-entries.

ABV has no tunable parameters.

FIS Tree can be used with a variable height of the FIS tree itself (a larger value saves memory but increases the number of memory accesses), and different algorithms to solve the range-lookup problem on each dimension. For the latter, our implementation can use a variety of search trees, some with a fixed branching factor, some with a different branching factor at each node. We have run a number of simulations, with the best results achieved using a FIS tree of depth 2 or 3, variable branching factor on the range lookup for the first dimension, and fixed branching on the second dimension.

In G-filter, we can configure the number m of partitions of each region, depending on the level and the

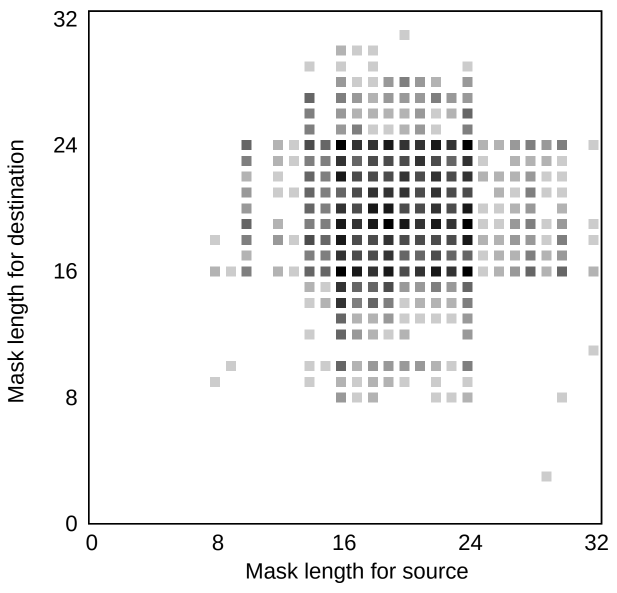

Fig. 4. Prefix length distribution (log scale).

Ruleset (size)	Alg.	Mem. usage	Cache acc.	Word acc.	Notes
juniper (210)	RFC	320K	7*	7*	hash lookups
	ABV	68K	66	67	
	FIS	21K	14	23	3-deep tree
	FIS	29K	12	21	2-deep tree
	G-filter	16K	11	44	
ipfw (238)	RFC	320K	7*	7*	hash lookups
	ABV	51K	66	67	
	FIS	23K	22	31	3-deep tree
	FIS	30K	20	29	2-deep tree
	G-filter	31K	17	63	
synth. (34K)	RFC	–	–	—	
	ABV	300M	66	99	
	FIS	2.2M	13	22	3-deep tree
	FIS	2.6M	10	16	2-deep tree
	G-filter	1.1M	5	42	m=256x256
	G-filter	5.0M	3	18	t=5
synth. (78K)	RFC	–	–	—	
	ABV	1015M	67	142	
	FIS	4.5M	15	24	3-deep tree
	FIS	6.4M	11	19	2-deep tree
	G-filter	2.2M	10	48	m=32x32
	G-filter	2.3M	6	42	m=256x256
	G-filter	67.8M	3	18	m=4Kx4K, t=5
synth. (200K)	RFC	–	–	—	
	ABV	–	–	—	
	FIS	9.5M	17	26	3-deep tree
	FIS	13.9M	12	21	2-deep tree
	G-filter	5.9M	10	51	m=32x32
	G-filter	9.4M	7	43	
synth. (540K)	RFC	–	–	—	
	ABV	–	–	—	
	FIS	20.5M	20	29	3-deep tree
	FIS	53.0M	13	22	2-deep tree
	G-filter	15.8M	12	55	m=8x8
synth. (1.3M)	RFC	–	–	—	
	ABV	–	–	—	
	FIS	44.3M	22	31	3-deep tree
	FIS	103.8M	14	23	2-deep tree
	G-filter	29.4M	12	66	
	G-filter	90.6M	8	42	m=4Kx4K

TABLE II
SUMMARY OF EXPERIMENTAL RESULTS FOR DIFFERENT ALGORITHMS, RULESET SIZES AND PARAMETERS.

number of dimensions, and the threshold t below which we store rules into arrays instead of performing the recursive partitioning. In all experiments, we use $m = 8^d$ for all levels after the first one. Unless otherwise specified, the first level is partitioned in $m = 1024^2$ regions, and the threshold for the use of arrays is $t = 13$ memory words.

E. Experimental results

The most significant experiments for all algorithms and data sets are summarised in Table II. The two small rulesets, derived from real firewall rulesets, are called *juniper* and *ipfw* with 210 and 238 rules, respectively. For the large rulesets, we have produced synthetic rule-set ranging from 34k to 1.3 million rules.

Small Rulesets: As it can be seen, for small rulesets RFC is the fastest algorithm (but with a warning – we only count the number of hash table accesses – the actual number of memory accesses might be larger if memory fills up), but it uses 5-10 times more memory than the other algorithms. For such small rulesets the memory overhead is not worrysome, though.

FIS and G-filter are on similar performance levels, in terms of both on memory usage and cache-line accesses (which is reasonably proportional to the actual memory access time). If we count the actual number of memory words accessed, G-filter appears to be worse, but this is an artifact of the use of arrays, widely used for small rulesets, and where each rule uses 2 or 3 words.

ABV tends to be largely worse than the others if we count cache-line accesses, mostly because the 1-bit tries used by the original implementation tend to be deep and make poor use of memory locality. The use of some kind of level-compressed tries might reduce the number of accesses to smaller values.

Large Rulesets: As the ruleset size increases, RFC and ABV start showing their severe scalability problems. In particular, RFC could not complete the data structure construction phase for any of the larger rulesets. In fact, already with a 4k ruleset, it starts using over 20MB of memory.

ABV shows a memory usage explosion already with the 34k ruleset, due to the need to store large lists of rules, not easy to compress, for each node of the tries.

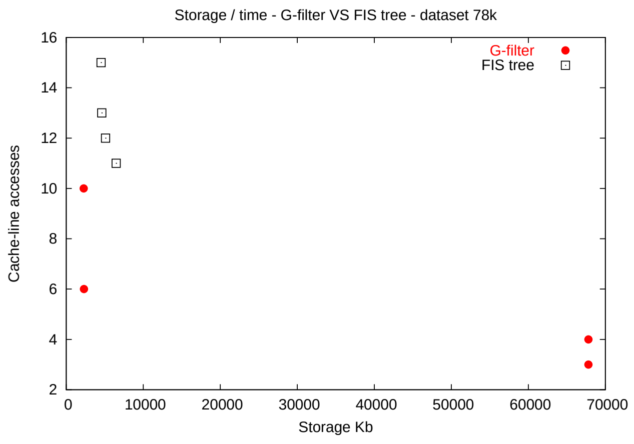

Fig. 5. G-filter VS FIS tree - ruleset 78k

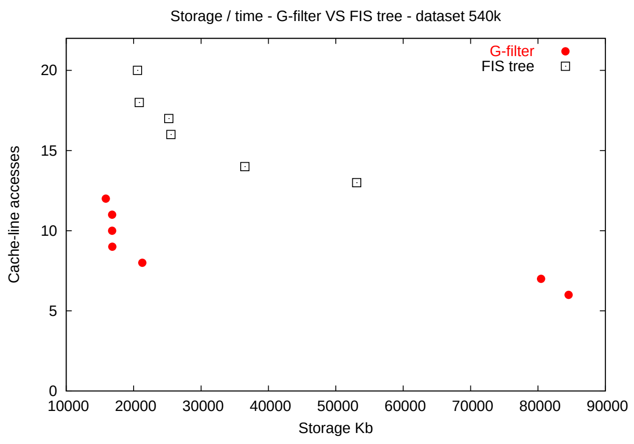

Fig. 6. G-filter VS FIS tree - ruleset 540k

FIS and G-filter are the only two algorithms that can cope with very large rulesets, while still using a reasonable amount of memory (30-40 bytes per rule in the best cases) and with rather interesting performance in terms of classification times. From our experiments, G-filter consistently and significantly outperforms FIS tree, by up to a factor of 2, whether we optimize the parameters for memory usage or for cache-line accesses.

To further extend the results in the Table, Figures 5, 6 and 7 show the space-time performance of FIS tree and G-filter for different values of the tunable parameters on the 78K, 540K and 1310K rulesets. As it can be seen, both algorithms can implement different space-time tradeoffs, but in general, the G-filter performance is always clearly better than the one of FIS Tree.

V. Conclusions and future work

We have presented a geometry-based algorithm for packet classification on $d-$dimensions that is suitable for large rulesets, but has reasonably good performance

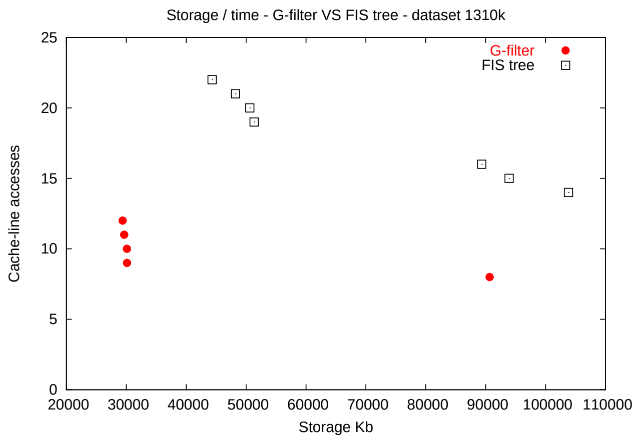

Fig. 7. G-filter VS FIS tree - ruleset 1310k

also on very small rulesets. On large rulesets, G-filter clearly outperforms the best proposal in the literature (FIS tree). Furthermore, its suitability to more than 2-dimension filtering makes it an interesting and practical candidate to the building of large $d-$dimensional packet classifiers.

The experiments presented in this paper are focused on 2-dimensional filters in order to compare G-filter with other approaches proposed in the literature. In the future we plan to run extensive experiments on the behaviour of our algorithm on large multi-dimensional rulesets.

References

[1] A. Feldmann and S. Muthukrishnan, “Tradeoffs for packet classification,” in *INFOCOM (3)*, 2000, pp. 1193–1202.

[2] Luigi Rizzo, “ipfw2 manual page,” http://www.freebsd.org/cgi/man.cgi?query=ipfw.

[3] Darren Reed, “Ipfilter web page,” http://www.phildev.net/ipf/.

[4] Daniel Hartmeier, Mike Franzen, Cedric Berger, Rayan McBride, and Can Erkin Acar, “Pf: The openbsd packet filter,” http://openbsd.org/faq/pf/.

[5] Gilbert A. Held, “Working with cisco access lists,” *Int. J. Netw. Manag.*, vol. 9, no. 3, pp. 151–154, 1999.

[6] “Juniper firewall filter configuration,” http://www.juniper.net/.

[7] D. Eppstein and S. Muthukrishnan, “Internet packet fileter management and rectangle geometry,” in *Proceedings of the 12th Annual Symposium on Discrete Algorithms*, New York, NY, USA, Jan. 2001, pp. 827–835, ACM Press.

[8] Marco Pellegrini, “Fast internet packet filtering on any number of attributes via multi-dimensional point stabbing,” Tech. Rep., IIT-CNR, Istituto di Informatica e Telematica del CNR, 2001.

[9] C. Matsumoto, “Cam vendors consider algorithmic alternatives,” in *EETimes*, May 2002.

[10] Florin Baboescu, Sumeet Singh, and George Varghese, “Packet classification for core routers: Is there an alternative to cams?,” in *INFOCOM*, 2003.

[11] Pankaj Gupta and Nick McKeown, “Packet classification using hierarchical intelligent cuttings,” in *Proc. Hot Interconnects VII*, 2000, pp. 34–41.

[12] Lili Qiu, George Varghese, and Subhash Suri, “Fast firewall implementations for software-based and hardware-based routers,” in *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 2001, pp. 344–345, ACM Press.

[13] Florin Baboescu and George Varghese, “Scalable packet classification,” in *Proceedings of INFOCOM 2001*. 2001, pp. 199–210, ACM Press.

[14] Pankaj Gupta and Nick McKeown, “Packet classification on multiple fields,” in *Proceedings of INFOCOM 1999*. 1999, pp. 147–160, ACM Press.

[15] P. Gupta and N. McKeown, “Algorithms for packet classification,” in *IEEE Network*, 2001, pp. 24–32, vol: 15:2, 2001.