Randomized Packet Filtering through Specialized Partitioning of Rulesets

Luca Abeni, Nicola Bonelli and Gregorio Procissi

***Abstract*—A key issue in high speed traffic processing is to immediately detect potentially interesting packets. At very high speed, this operation is particularly crucial as filtering packets *close to the wire* relieves real applications from handling large volumes of (uninteresting) data. This paper proposes a fast and randomized approach to packet filtering based on partitioning rule databases for their storage in fast and compact Bloom filters that can be placed in fast cache memory. Database partitioning is obtained by a specially tailored clustering algorithm and the results show that even large rulesets can be divided into a limited number of partitions and accommodated in reasonably small Bloom filters.**

***Index Terms*—Packet filtering, Bloom filters, Rules database, Partition, Clustering**

I. Introduction

THE huge amount of data exchanged in todays' Internet, and the fast and continuous proliferation of new services and cyber attacks require data monitoring and processing operations to be quickly responsive and to run *on the live data directly*. In addition, new technologies – such as network virtualization – arise similar requirements, as high speed packet forwarding in huge and high–loaded data centers calls for fast and efficient data handling and steering mechanisms.

In all the above cases, efficient data processing requires a first data–reduction stage of *filtering* to immediately recognize the packets of interest (namely, either the packets to be collected in a monitoring/security application, or the packets to be forwarded in a virtual network). Packet filtering is a very special case of packet *classification* in which the results are only *yes* (packet allowed) or *no* (packet is dropped). Packet classification, instead, is a more complex task that involves searching for the *best matching rule* in the database. As such, it requires exact algorithms and very large data structures that hardly fit into small memory caches.

Also, it is worth reminding that packet filtering is typically used as a data–reduction stage for a second stage of processing in which a second check on the data coming at a significantly reduced rate can still be performed. Hence, a small – but controlled – number of *false positives* can generally be tolerated (false negatives, instead, must be avoided to refrain from losing packets). This work exploits such a property by presenting a novel randomized approach for packet filtering, which allows to process packets from a high–speed link by trading few false positives in favour of high performance. To achieve the desired trade–off between false positives and performance we compact the state information in a Bloom filter [1]. The resulting small memory footprint makes it possible to take advantage of small but fast cache memories of modern computer architectures to achieve high performance.

The use of Bloom filters for packet filtering is not new, though not very widespread. Deri [2] proposed the use of a Bloom filtering stage for fully specified rules with limited support of wildcards. In [3] and [4], Bloom filters are used as part of more complex tasks such as packet lookup and classifications. Recently, [5] proposes an approach to packet classification in which Bloom filters are used for all involved header fields as well as to combine the results for tuple pruning before accessing an off–chip hash table. In this paper, instead, we aim at using a single Bloom filter to provide a full–fledged technique for packet filtering for nearly general rule types (i.e. with support of ranges of values of header fields).

L. Abeni is with DISI, University of Trento, Via Sommarive 5, POVO 38123 Trento, Italy e-mail: luca.abeni@unitn.it,

N. Bonelli and G. Procissi are with the Department of Information Engineering – University of Pisa, and CNIT, Via G. Caruso 16, 56122, Pisa, Italy e-mail: nicola.bonelli@cnit.it, g.procissi@iet.unipi.it

II. Filtering Taxonomy

Packet filtering is the network function whose result is either accepting or discarding packets according to a set of *rules* (ruleset or rules database). Typically, rules specify the values that packet header fields (or part of them) must match for the packet to be accepted.

More formally, a header is modelled as a sequence of K fields, where the i^{th} field is composed by ν_i bits. A filtering rule is a tuple of pairs H_i/p_i:

$$R = (H_1/p_1, H_2/p_2, \dots, H_K/p_K) \tag{1}$$

meaning that, for any packet to be checked, the first p_i bits of its i^{th} header field are compared to the the first p_i bits of H_i ($p_i, 0 \leq p_i \leq \nu_i$). If a match occurs for all $1 \leq i \leq K$ header fields, then the packet is accepted. Otherwise it gets dropped. The tuple of the bitwise prefix lengths of a rule R defines its *signature* σ, namely:

$$\sigma(R) = (p_1, p_2, \dots, p_K) \tag{2}$$

Notice that, while a rule R is associated with one and only one signature $\sigma(R)$, multiple rules may have the same signature.

Ideally, a very effective approach to packet filtering would then be to "expand" all of the rules in the database against the limiting signature $(\nu_1, \nu_2, \dots, \nu_K)$ generating a list of all the accepted header values, and to insert such values (the "expanded rules") into a single Bloom filter. If, on one hand, this approach would achieve $O(1)$ membership lookup complexity, on the other hand the number of entries generated by such an expansion would be far too large to be accommodated into a reasonably small Bloom filter that can fit in a standard cache memory. While the philosophy of this approach can be kept, a more elaborated strategy for rules expansion is needed to maintain low complexity and meet memory constraints.

Let the rules *database* $\mathcal{R}$ (ruleset):

$$\mathcal{R} = \{R_1, R_2, \dots R_L\} \tag{3}$$

have cardinality L and let $\mathcal{S} = \{\sigma_1, \sigma_2, \dots \sigma_P\}$ be the set of signatures that occur in the database. By denoting with $\mathcal{R}_{\sigma_i}$ the subset of $\mathcal{R}$ containing rules with the same signature σ_i, a natural partition of the set $\mathcal{R}$ is given by:

$$\mathcal{R} = \bigcup_{i=1}^{P} \mathcal{R}_{\sigma_i} \text{ with } \mathcal{R}_{\sigma_i} \bigcap \mathcal{R}_{\sigma_j} = \emptyset \; i \neq j = 1, 2, \dots, P \tag{4}$$

Notice that the original ruleset $\mathcal{R}$ may contain redundant rules: the partition (4) allows to reconstruct $\mathcal{R}$ exactly as it is, including all redundancies possibly contained therein.

Let us now consider the set of rules $\mathcal{R}_\sigma$ associated with signature σ. Each element of this set can be represented in terms of any other signature ξ having all components bigger than or equal to those of σ, at the cost of exploding the number of rules by a factor $\epsilon(\sigma, \xi)$ (*explosion factor*) given by 2 to the power of the Manhattan distance between the signatures σ and ξ, that is:

$$\epsilon(\sigma, \xi) = \prod_{i=1}^{K} 2^{\left|p_i^{(\sigma)} - p_i^{(\xi)}\right|} = 2^{\sum_{i=1}^{K}\left|p_i^{(\sigma)} - p_i^{(\xi)}\right|} \tag{5}$$

The cardinality of the "expanded" set will then be:

$$|\mathcal{R}_\xi| = |\mathcal{R}_\sigma| \, \epsilon(\sigma, \xi) \tag{6}$$

Finally, we define *minimum common signature* (mcs) of a set of M signatures $\Sigma = \{\sigma_1, \sigma_2, \dots \sigma_M\}$ the *minimum* signature having all the components bigger to or equal than all the components of the signatures of Σ, i.e.:

$$\text{mcs}\,(\Sigma) = \left(\max_{1 \leq i \leq M} p_1^{(\sigma_i)}, \dots, \max_{1 \leq i \leq M} p_K^{(\sigma_i)} \right) \tag{7}$$

Figure 1 shows a 2D example with several subsets of signatures referring to source and destination IP addresses, together with their associated minimum common signature on the upper–right corner. Notice that, given a generic set of signatures Σ, $\text{mcs}\,(\Sigma)$ does not necessarily belong to Σ as well.

More in general, any set of rules $\mathcal{R}$ partitioned according to the set of signatures Σ that occur in $\mathcal{R}$ (i.e., $\mathcal{R} = \bigcup_{\sigma \in \Sigma} \mathcal{R}_\sigma$), can be exploded according to the minimum common signature of Σ. The resulting number of rules after explosion is:

$$\left|\mathcal{R}_{\text{mcs}(\Sigma)}\right| = \sum_{\sigma \in \Sigma} |\mathcal{R}_\sigma| \, \epsilon\left(\sigma, \text{mcs}\,(\Sigma)\right) \tag{8}$$

III. The filtering problem

As already explained, our goal is to fit the whole ruleset $\mathcal{D}$ in a single standard Bloom filter small enough to fit in the CPU cache (in order to allow fast accesses). To this end, the ruleset is partitioned into a limited number N of subsets,

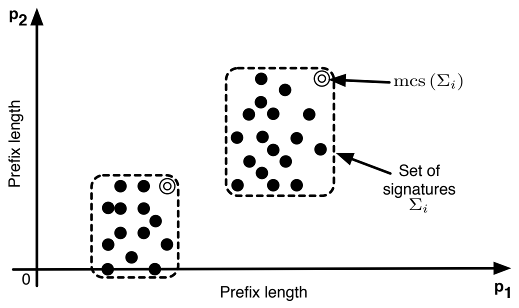

Fig. 1. 2D example of database partition with maximum mask.

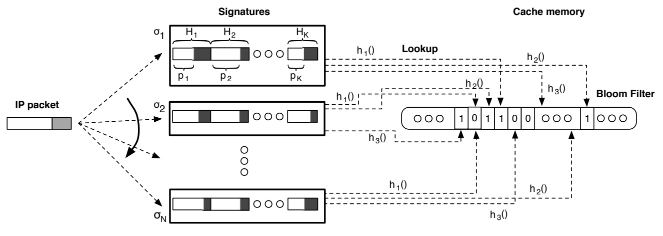

Fig. 2. Signatures masking and Bloom filter lookup ($k = 3$ hash functions)

and all rules belonging to the same partition are expanded (if needed) and expressed according to the minimum common signature of the set of signatures associated with the partition. At the end of this process, N distinct subsets of rules are ready to be inserted into the Bloom filter, with all rules belonging to the same subset being expressed according to a single common signature. Notice that rules are inserted in the Bloom filter in a flat way, regardless of the subset they belong to (i.e. regardless of their signature). Practically speaking, this only requires the use of hash functions that accept variable length bitmaps as arguments (as signatures from different subsets of rules have different lengths). The role of subsets becomes crucial at lookup time, instead. As graphically depicted in figure 2, upon each packet arrival: i) its K header fields are progressively masked according to the minimum common signatures of the N partitions (i.e., subsets), and ii) the resulting K-uple is checked in the Bloom filter until either a match occurs (in this case the packet is forwarded to the next stage) or the end of signatures is reached (the packet is filtered out).

Since the number of memory accesses performed by the filtering mechanism is proportional to the number of partitions N, the first goal of the proposed approach is to *minimize* N. On the other side, decreasing N increases the number of expanded rules and the size of the Bloom filter, with the risk of preventing it to fit in the CPU cache. Therefore, the resulting problem is a *constrained minimization*: find the minimum value of N such that the memory footprint of the resulting Bloom filter is smaller than the CPU cache size.

The problem of partitioning a set of signatures into a given number of subsets is a typical problem of *clustering*. Hence, the problem of partitioning the ruleset (and, in turn, the set of signatures) can be translated into a problem of signature clustering where the *centroid* of each cluster is set to the cluster's minimum common signature. In addition, each cluster is associated with a *cost* given by the number of expanded rules induced by the corresponding cluster centroid.

For a standard Bloom filter, given its size m, the number k of hash functions used to index it, and the tolerated false positive probability f, it is well known [6] that the number n_{max} of elements that can be accommodated is:

$$n_{max} = -\left(\frac{m}{k}\right)\log\left(1 - f^{1/k}\right) \tag{9}$$

Thus, the total number of expanded rules (that is, the sum of the costs of all clusters) must not exceed n_{max} (the *cost threshold*).

```
Function HyperMerge(points, cost_threshold)
  $n \leftarrow 0$;
  repeat
      $n \leftarrow n + 1$;
      $partitions \leftarrow do\_partitioning(points, n)$;
      $cost \leftarrow compute\_cost(partitions)$;
  until $cost \leq cost\_threshold$;
  return partitions;
```

This constrained minimization problem can be solved by using the *HyperMerge* algorithm, an iterative signature-aware algorithm that receives the set of signatures Σ (modelled as a set of n-dimensional points) and the cost threshold n_{max} as inputs, and produces the partition $\{\mathcal{P}_i\}$ as output. The algorithm tries to group the n-dimensional points (representing the signatures) in an increasing number N of partitions, by using the `do_partitioning()` function *until the cost is smaller than the threshold* n_{max}.

IV. The Partitioning Algorithm

While HyperMerge may rely on any kind of clustering algorithm (the actual "content" of the `do_partitioning()` function), the overall effectiveness of the algorithm (i.e., its ability of finding smaller values of N for which the memory constraint is respected) depends on the adopted clustering algorithm. Although a generic clustering algorithm (for example, K-Means [7]) can be used for partitioning, a novel and specialized K-Centroid type algorithm, named MinExp, has been developed. MinExp uses the minimum common signature of a partition as a centroid and the explosion factor (5) as a distance measure.

Function `MinExpPartitioning` describes this greedy optimization algorithm that starts from a random distribution of the points in N partitions and iteratively checks if the total cost can be reduced by moving a point to a different partition. When no point can be further moved, the algorithm stops. Since MinExp is a greedy algorithm (at each step it performs a locally optimal move), it is not guaranteed to find an absolute minimum (but only a local minimum). As a consequence, the initialization step is very important and heavily affects the final result. In particular, if partitions are initialized by using the K-Means algorithm (i.e., starting from a random distribution of the points in the partitions, then running K-Means, and then running MinExp on the resulting partitions) the final

```
Function MinExpPartitioning(points, n)
  repeat
      update centroids; $moved \leftarrow false$;
      foreach $p$ in $points$ do
          $oc = compute\_cost(points, partitions)$;
          $old\_partiton \leftarrow partition(p)$;
          $new\_partition \leftarrow partiton(p)$;
          foreach $partition$ in $partitions$ do
              move $p$ to $partition$; update centroids;
              $nc = compute\_cost(points, partitions)$;
              if $nc < oc$ then
                  $oc \leftarrow nc$; $new\_partition \leftarrow partition$;
              if $old\_partition \neq new\_partition$ then
                  $moved \leftarrow true$;
  until $moved == false$;
  return partitions;
```

TABLE I
Performance of K-Means, MinExp, and K-MinExp.

	Avg	Std Dev	99% conf	Size
K-Means	11.800000	2.111871	3.651425	
MinExp	9.700000	0.842615	1.456881	100
K-MinExp	8.300000	0.842615	1.456881	
K-Means	27.500000	5.500000	9.509500	
MinExp	31.800000	4.237924	7.327371	500
K-MinExp	14.050000	0.864581	1.494860	
K-Means	42.600000	6.583312	11.382547	
MinExp	56.150000	6.373971	11.020595	1000
K-MinExp	18.150000	1.194780	2.065775	

number of clusters is significantly reduced (see Section V) and the execution time greatly decreases. In the following, the combined use of K-Means (for initialization) and MinExp algorithms is referred to as K-MinExp.

V. Performance Evaluation

The performance of HyperMerge has been evaluated by comparing the clustering algorithm K-MinExp to both K-Means and MinExp alone. Remember that the number of memory accesses used for filtering is proportional to N, hence N can be used as a performance metric. We assumed an 8 MB Bloom filter (a common CPU cache size in todays' commodity architectures) equipped with $k = 4$ hash functions that, by equation (9), can handle around 1.77 Millions rules with false probability of 10^{-4}.

In a first set of experiments, the performance of the three algorithms has been compared by using different rulesets randomly generated by ClasshBench [8]. To assess scalability, different kinds of rulesets (ranging from 100 rules to 1000 rules) have been generated, and HyperMerge has been used to partition the rules and obtain the minimum number of generated partitions. Each experiment has been repeated 20 times (using 20 different random rulesets for each size). Table I shows the results obtained for sets of 100, 500, and 1000 rules. While it is quite evident that K-MinExp always out-

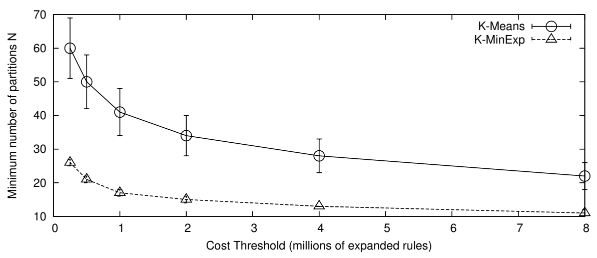

Fig. 3. Performance of K-Means vs K-MinExp as a function of the cost threshold n_{max}.

performs MinExp and K-Means, the results of the comparison between MinExp and K-Means are less obvious: for smaller rulesets (100 rules) MinExp performs consistently better than K-Means, but for larger sets K-Means often outperforms MinExp (although in some cases MinExp performs better than K-Means). These results indicate that without a proper initialization MinExp does not scale well with the ruleset size. However, when K-Means is used to initialize MinExp the resulting algorithm (K-MinExp) performs very well and scales properly.

In the next set of experiments, the impact of the cost threshold (used to stop the HyperMerge algorithm - see the `until` condition in Algorithm `HyperMerge`) is evaluated. Figure 3 compares the performance of K-Means and K-MinExp (in terms of minimum number of partitions generated by HyperMerge) for different values of thresholds. Each experiment is based on a ruleset of size 1000 (1000 rules randomly generated with ClasshBench) and has been repeated 100 times (by generating 100 rulesets per experiment). The figure plots the mean and the standard deviation on the 100 runs. Again, note that K-MinExp constantly outperforms K-Means. Moreover, it is interesting to notice that K-Means results are subject to noticeable variations from run to run (the standard deviation is significant), while K-MinExp results are pretty consistent (the standard deviation is always pretty small).

Repeating the experiments multiple times with different numbers of rules, it has been observed that the number N of partitions generated by K-MinExp is $O(\log(L))$ (where L is the ruleset size). Therefore, the number of memory accesses of HyperMerge is $O(\log(L))$. Comparing this result to well known algorithms from literature (see [9], Table 8), it can be seen that all the algorithms requiring a number of memory accesses smaller than $O(L)$ have a large memory footprint (and their data structures do not fit in a CPU cache), while the algorithms using data structures that can fit in the CPU cache require a number of memory accesses that is at least $O(L)$. HyperMerge is therefore the only one providing a good trade-off between time and space complexity.

After testing the performance of MinExp on synthetic rulesets, some more realistic experiments have been performed by considering real rulesets taken from [8]. In particular, 3 rulesets have been considered: `FW`, `ACL`, and `IPC` [8][1]. The

[1]The rulesets are downloadable from http://www.arl.wustl.edu/~hs1/PClassEval.html#3._Filter_Sets.

TABLE II
NUMBER OF PARTITIONS OBTAINED USING REAL RULESETS.

	FW	ACL	IPC
K-Means	16	14	189
MinExp	14	30	273
K-MinExp	11	14	40

results on the number of partitions obtained are shown in Table II. Notice that once again K-MinExp provides the best performance (for the `ACL` ruleset only K-Means compares to it). Also notice that in the `IPC` case the performance difference between K-MinExp and the other two algorithms is very relevant.

VI. Conclusions and Future Works

The paper presents a randomized approach to fast packet filtering for high speed data processing. The main idea behind the proposed technique is to fit the whole filtering ruleset in a single Bloom filter for fast lookup upon a proper partitioning of the set. The results prove that even large rule databases (with different statistical properties) can be accommodated in a reasonably small Bloom filter which fits in the CPU cache while achieving $O(\log L)$ time complexity. The types of rules here addressed include any kind of header values. Hence, we plan to extended the presented approach to include the basic rules used in Openflow lookup operations.

Acknowledgment

This work was partially supported by the Italian Ministry of University and Research funded projects SFINGI and GreenNet.

References

[1] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, July 1970.

[2] L. Deri, "High-speed dynamic packet filtering," *J. Netw. Syst. Manage.*, vol. 15, no. 3, pp. 401–415, Sep. 2007. [Online]. Available: http://dx.doi.org/10.1007/s10922-007-9070-0

[3] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '03. New York, NY, USA: ACM, 2003, pp. 201–212. [Online]. Available: http://doi.acm.org/10.1145/863955.863979

[4] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood, "Fast packet classification using bloom filters," in *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, ser. ANCS '06. New York, NY, USA: ACM, 2006, pp. 61–70. [Online]. Available: http://doi.acm.org/10.1145/1185347.1185356

[5] H. Lim and S. Y. Kim, "Tuple pruning using bloom filters for packet classification," *IEEE Micro*, vol. 30, no. 3, pp. 48–59, 2010.

[6] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 254–265, 1998.

[7] E. W. Forgy, "Cluster analysis of multivariate data: efficiency vs interpretability of classifications," *Biometrics*, vol. 21, pp. 768–769, 1965.

[8] D. Taylor and J. Turner, "Classbench: A packet classification benchmark," *Networking, IEEE/ACM Transactions on*, vol. 15, no. 3, pp. 499–511, june 2007.

[9] P. Gupta and N. McKeown, "Algorithms for packet classification," *Netwrk. Mag. of Global Internetwkg.*, vol. 15, no. 2, pp. 24–32, Mar. 2001. [Online]. Available: http://dx.doi.org/10.1109/65.912717