Scalable Remote Firewalls

Michael Paddon, Philip Hawkes, Greg Rose

Qualcomm

<mwp@qualcomm.com>, <phawkes@qualcomm.com>, <ggr@qualcomm.com>

ABSTRACT

There is a need for scalable firewalls, that may be dynamically configured by the network nodes that they service. While modern stateful filtering techniques are good at dealing with established traffic flows, the scalable classification of other packets is a less researched problem. A novel method for scalable packet classification on arbitrary criteria is proposed that addresses this requirement. The classifier supports dynamically updatable policies comprised of sequence insensitive rules. Experimental data is presented that demonstrates efficient and scalable performance with large policies. The classifier is therefore suitable for use in scalable remote firewalls.

1. Rationale

Traditionally, firewalls have been used to protect relatively small networks from the world at large. Over the last few years, as "personal firewalls" have become common, the trend of protecting smaller domains with finer grained policy has reached its logical conclusion. The security benefits of this evolution have generally been positive: improving the configurability, utility and (in the case of mobile devices) portability of firewalls.

A fundamental economic assumption underlying the "every node is a firewall'' model is that the cost of delivery of unwanted packets is negligible. This assumption is not always true. In general, any link for which demand exceeds (or is likely to exceed) available bandwidth exhibits non-trivial packet delivery costs. We call these *expensive* links. Typical examples might be low speed or heavily congested channels.

One interesting class of expensive links is those constrained by intrinsic physical properties. For instance, wireless transports utilise radio spectrum resources that are strictly limited in availability. One can always lay more fibre between two points, but wireless bandwidth is far less elastic. As a consequence, wireless spectrum is regarded as a precious resource, as attested by high licensing costs in most jurisdictions.

There exists, therefore, an economic imperative to prevent unwanted packets from traversing expensive links. This implies a requirement for centralised firewalls at link ingress points, that enforce desired traffic policy. Because an expensive link may provide connectivity to an unbounded number of nodes, such firewalls must scale well and gracefully support large populations.

In order to be effective, these firewalls need not be perfect; any significant reduction in unwanted traffic is a net gain. However, the more precisely the enforced policy fits the actual traffic requirements of legitimate node population, the more effective it will be. Therefore, these firewalls should permit remote *ad hoc* updates to policy (from authorised sources).

One common class of firewall is the packet filter: a technology which passes or blocks packets, but otherwise leaves traffic flows unperturbed. At the core of every packet filters is a mechanism which classifies packets according to a supplied policy. This paper describes a novel packet classification technique that is highly scalable and supports dynamic policy updates.

2. Problem Statement

Existing stateful packet filters (such as OpenBSD's *pf*[5]) possess scalable mechanisms for processing packets that belong to established traffic flows. Packets that do not belong to an established flow are classified according to a policy which is expressed as a set of rules. Rules are generally processed in sequence in order to assess each packet. This approach exhibits $O(N)$ complexity, relative to the size of the rule set.

Some packet classifiers employ optimisation techniques to their rule sets in order to speed up packet processing. Facilities for early termination of rule processing under specified circumstances are common. A more sophisticated example is *pf*'s *skip-steps*, which enable predictive skipping when contiguous rule blocks could never match a packet. The effect is much like constructing a tree. Such techniques can be very effective if the rule set is highly ordered and exhibits strong commonality in rule criteria. In a highly dynamic

environment, where there are ongoing incremental updates to the rule set, this is not generally true.

Traditionally, classifier rule sets tend to be quite static in nature, and are often updated via a manual process. This is more a matter of convention than technical constraint. Nevertheless, since extant classifiers typically exhibit sequence dependent behaviour, it is generally difficult to insert and remove arbitrary rules from a policy without unwanted or unintended side effects.

Nodes protected by a centralised packet filter may wish to extend service (typically by listening for packets that initiate a flow) at any time. Similarly, they may wish to retract heretofore offered services. This is consistent with the internet end-to-end model. If the maximum number of unwanted packets are to be blocked while allowing *ad hoc* service extension and retraction, then the filtering policy must be dynamically updated by nodes as changes occur.[1]

There exists, therefore, a need for a packet classifier that expresses policy using sequence independent rules. Rules must be able to be dynamically inserted into, or deleted from, the policy at any time. Classification performance must be scalable and exhibit fundamentally better performance than $O(N)$.

3. Prism Packet Classification

Our packet classification technique operates by representing each packet as a point and each rule as a prism in n-dimensional space. Fast matching of packets against prisms is achieved by using a spatially indexed data structure.

3.1 Packet Features

We consider a packet to possess a predetermined set of n interesting features for the purposes of classification. Each feature must be defined so that it is representable by a number that falls within a predetermined range. Features may be represented by floating point numbers, but are most often integral in nature. Distinct features need not be orthogonal.

For instance, the source and destination addresses of an IPv4 packet may be used directly as features as they are each representable by an integer with a predetermined range of $[0,2^{32}-1]$ ($[0,2^{128}-1]$ in the case of IPv6). The upper layer protocol number is another example of a typical feature, being an integer in the range of $[0,255]$. In general, any information in a packet may be used either directly as a feature, or to algorithmically construct a feature. In either case, we say that the information generates the feature.

Information that may or may not be present in the packet may also be used to generate features. Typical examples of such information would be optional data (such as IPv4 header options or IPv6 optional headers). Information from the packet payload may also be used to generate features, such as fields from encapsulated upper layer protocol headers. Typical examples of such information would be TCP or UDP ports numbers, and ICMP types and codes. When such optional information is not present, the generated feature must take on a predetermined value; in other words, even if the information is not present, the feature is still defined.

Features may also be generated using information recalled from previous packets. In other words, feature generation may be stateful.

3.2 Feature Vectors and Prisms

Each packet may be represented as a fixed length vector ν, consisting of n feature values μ:

$$\nu = \langle \mu_1, \ldots, \mu_n \rangle$$

Each ν describes a point in an n-dimensional affine feature space. We will call this space Φ. The precise set of features chosen as the axes of Φ depends on the classifying application's requirements.

An axis-aligned n-dimensional cuboid ψ in Φ may be defined by specifying a contiguous sub-range for each axis:

$$\psi = \langle [\mu_{low_1}, \mu_{high_1}], \ldots, [\mu_{low_n}, \mu_{high_n}] \rangle$$

We will call these cuboids *feature prisms*. A feature prism may be thought of as a geometrically coherent set of packet classification criteria.

Prism ψ *encloses* vector ν if and only if:

$$\forall \mu_i \in \nu \text{ and } [\mu_{low_i}, \mu_{high_i}] \in \psi,$$

$$\mu_{low_i} \leq \mu_i \leq \mu_{high_i}$$

3.3 Packet Classification

Let Ψ be an arbitrary set of feature prisms, forming the rule set of a packet classifier.

Vector ν *matches* Ψ if and only if:

1. The filter should also have a mechanism (such as *keep-alives*) to discover when a node departs the network abruptly, so that obsolete rules can be elided from the policy in a timely fashion.

$$\exists \psi \in \Psi, \text{ such that } \psi \text{ encloses } \nu$$

A packet may, therefore, be binary classified relative to Ψ. The semantics of ν matching Ψ depends on the application. Matching may mean that the associated packet is permitted through the filter. In this case Ψ represents a positive rule set. If Ψ is interpreted as a negative rule set, then a match would result in the packet being blocked. More complex classification is possible by matching ν against a sequence, or even a decision tree, of distinct Ψ.

3.4 Fast Packet Matching

The efficient determination of whether a point in n-dimensions is enclosed by one or more regions is a well studied problem, with an extensive literature. Such techniques are generally known as spatial access methods (SAMs). One particularly successful class of SAM is the R-tree[1] and its many variants such as R+-trees[2] and R*-trees[3]. A good general survey of R-trees and related data structures is given by Manolopoulos *et al*[4].

R-trees are an extension of the well known B+-tree data structure, in which the keys are multidimensional rectangles. Interior nodes hold the minimum bounding rectangle (MBR) for each child. Classic R-trees and R*-trees allow MBRs to overlap, reducing tree size at the cost of potentially more expensive queries (as multiple branches of the tree may need to be traversed). R+-trees, on the other hand, guarantee disjoint MBRs, which may increase tree size (as keys may need to be stored in more than one leaf node). R*-trees are generally regarded as the best performing of the R-tree family. R-trees are dynamic data structures, so data may be inserted and deleted at any time.

A classifier rule set Ψ may be represented by an R-tree whose leaf MBRs are isomorphic with Ψ's elements. Efficient packet matching may then be achieved via a *point query* on the tree, which recursively searches nodes whose MBRs enclose the desired point until any matching prisms are found at the leaves. For the purpose of classification, query traversal may be terminated as soon as the first enclosing prism is detected.

A key measure of the performance of a point query is how many nodes were traversed to satisfy the query. Predicting the performance of R-Trees in practice is difficult, and there a few analytical results in the literature. The worst case complexity is $O(\tau)$, where τ is the number of nodes in the tree. Best case is $O(\log_d \tau)$ where d is the maximum degree of the nodes and the tree is fully populated. The actual performance is highly dependent on the contents of the tree.

4. Experimental Data

Let σ be the number of prisms and τ the number of nodes in a given R-tree. Let υ be the number of nodes traversed by a specific point query. We observe that $0 \le \upsilon \le \tau$, since the root node itself may never be visited if its MBR does not enclose the point.

We define a performance metric ω for ρ point queries as:

$$\omega = \frac{\sum_{i=1}^{\rho} \upsilon_i}{\rho\tau}$$

Note that $0 \le \omega \le 1$. Since $\sigma \propto \tau$, ω acts as a normalised approximation of the proportion of Ψ searched to satisfy the query.

Our experimental method consists of the algorithm:

1. Construct a R*-tree by inserting σ randomly generated "typical" prisms. Count the total number of node reads and writes required to build the tree.
2. Perform σ point queries, using a random point from inside each prism.
3. Perform σ point queries, using randomly generated "typical" vectors.
4. Calculate ω for all point queries that successfully matched a prism.
5. Calculate ω for all point queries that failed to match any prism.

We used Hadjieleftheriou's free Spatial Index Library[6] as our R*-tree implementation. The default node splitting algorithm parameters were used, including a fill factor of 0.7, a split distribution factor of 0.4 and a reinsert factor of 0.3.

4.1 Typical Prisms and Vectors

Our experimental feature space, Ψ, is chosen to contain seven typical packet classification criteria:

- source address (32 bits)
- destination address (32 bits)
- layer protocol (8 bits)
- ICMP type (8 bits) [ICMP only]
- ICMP code (8 bits) [ICMP only]
- source port (16 bits) [TCP and UDP only]
- destination port (16 bits) [TCP and UDP only]

When constructing a feature vector, features that are not present in a packet are defined as zero.

A typical feature prism is intended to represent the type of rule that a node offering a service would generate in order to allow access to that service. Some services are offered to the world at large, and others are constrained in terms of source restrictions. Typical prisms are defined to be:

- source address (probability): all addresses (0.8), a random 24 bit CIDR block (0.1) or a random address (0.1)
- destination address: a random address in the 10.0.0.0/8 block
- upper layer protocol (probability): TCP (0.4), UDP (0.4), ICMP (0.1) or a random protocol (0.1)
- ICMP type: a random type [0 if not ICMP]
- ICMP code: a random code range [0 if not ICMP]
- source port (probability): all ports (0.8), ports 0 to 1023 (0.1) or a random port (0.1) [0 if not TCP or UDP]
- destination port: a random port [0 if not TCP or UDP]

Prisms were constructed with edges extending 0.5 units beyond the minima and maxima of each feature range.

A typical feature vector is intended to represent the type of packet that might be intercepted by a packet filter:

- source address: a random address
- destination address: a random address
- upper layer protocol (probability): TCP (0.4), UDP (0.4), ICMP (0.1) or a random protocol (0.1)
- ICMP type: a random type [0 if not ICMP]
- ICMP code: a random code [0 if not ICMP]
- source port: a random port [0 if not TCP or UDP]
- destination port: a random port [0 if not TCP or UDP]

4.2 Insertion Performance

Trees containing varying numbers of prisms were built, for several values of the maximum node degree d. The total number of node reads and node writes were observed at the completion of each tree's construction. Total accesses were defined as the sum of the reads and writes.

σ	reads	writes	accesses
10^2	299	191	490
10^3	5428	2685	8113
10^4	76821	33131	109952
10^5	979299	383959	1363258
10^6	11895335	4433011	16328346

$d = 16$

σ	reads	writes	accesses
10^2	219	139	358
10^3	3598	1876	5474
10^4	47809	20749	68558
10^5	650885	237571	888456
10^6	8109362	2669052	10778414

$d = 32$

σ	reads	writes	accesses
10^2	139	107	246
10^3	2874	1582	4456
10^4	42136	17745	59881
10^5	479027	183936	662963
10^6	6319883	2001566	8321449

$d = 64$

Recall that the amortised complexity of classic B-tree insertion is $O(\log N)$. Our data suggests that the insertion cost (in accesses) for σ prisms into our R*-Tree is proportional to $\sigma \log \sigma$. In other words, a single insertion is $O(\log \sigma)$.[2] This makes sense; R*-tree insertion, although more expensive due to a forced reinsertion strategy, is algorithmically similar to B-tree insertion. Note that the observed costs are *not* necessarily worst case behaviour. Nevertheless, in practice, insertion appears to scale well.

The insertion cost becomes cheaper as the maximum node degree is increased. However, large values of d create more CPU load in terms of node searching and splitting, and it is important to keep d below the threshold where such costs begin to dominate. Our point query results (below) suggest that the optimal d is less than 64.

4.3 Point Query Performance

Trees containing varying numbers of prisms were built, for several values of the maximum

2. Deletion cost is expected to be similar to that of insertion.

node degree d. The tree was then subjected to 2σ point queries, and the metric ω was calculated for successful and unsuccessful matches.

σ	τ	$\omega_{matched}$	$\omega_{unmatched}$
10^2	10	2.00×10^{-1}	0
10^3	103	5.30×10^{-2}	1.46×10^{-4}
10^4	1023	1.09×10^{-2}	3.99×10^{-5}
10^5	10235	1.60×10^{-3}	5.83×10^{-6}
10^6	102982	3.90×10^{-4}	1.45×10^{-6}

$d = 16$

σ	τ	$\omega_{matched}$	$\omega_{unmatched}$
10^2	5	4.00×10^{-1}	0
10^3	49	6.12×10^{-2}	1.84×10^{-4}
10^4	486	6.23×10^{-3}	2.35×10^{-5}
10^5	4736	1.03×10^{-3}	3.73×10^{-6}
10^6	47092	1.76×10^{-4}	6.54×10^{-7}

$d = 32$

σ	τ	$\omega_{matched}$	$\omega_{unmatched}$
10^2	3	6.67×10^{-1}	0
10^3	23	8.70×10^{-2}	2.61×10^{-4}
10^4	233	1.29×10^{-2}	4.72×10^{-5}
10^5	2304	2.12×10^{-3}	7.78×10^{-6}
10^6	22904	2.73×10^{-4}	1.03×10^{-6}

$d = 64$

Our data suggests that ω is roughly proportional to $\frac{1}{\sigma}$. It is also useful to examine the absolute number of nodes visited for a particular scenario. The following table shows the mean υ values for point queries when $d = 32$:

σ	τ	$\upsilon_{matched}$	$\upsilon_{unmatched}$
10^2	5	2.00	0
10^3	49	3.00	9.02×10^{-3}
10^4	486	3.03	1.14×10^{-2}
10^5	4736	4.88	1.77×10^{-2}
10^6	47092	8.29	3.08×10^{-2}

This is suggestive of a slightly worse than $O(\log\sigma)$ cost in terms of node visits for matching queries (in fact, our lower cost bound is $O(\log\sigma)$). We can see that, when $\sigma = 10^6$, an average point query visited 8.29 and 0.0308 nodes for successful and unsuccessful matches respectively. Even with a million rules, the classifier is capable of efficient operation.

We also note that non-matching queries are substantially cheaper than matching queries. This is highly desirable for a packet filter whose purpose, after all, is to reject spurious traffic as cheaply as possible.

Our results also show that for $\sigma = 10^6$, we see a minimal ω for both matched and unmatched packets when $d = 32$. This suggests that there is an optimal value for d for any expected σ, which allows trees to be tuned for their expected load. Even in the absence of such tuning, however, performance remains acceptable.

5. Conclusions

The proposed mechanism exhibits excellent scalability for both dynamic rule updates and packet classification. Non matching packets are assessed particularly quickly, which is a highly desirable characteristic. Therefore, this classifier is suitable for the creation of high performance packet filters that enforce policies with exceptionally large numbers of rules.

The classifier rules are sequence insensitive, allowing multiple parties to independently insert and remove rules without unwanted or unexpected interactions. This makes the mechanism eminently suitable to a highly dynamic environment, where policy is an amalgam of rules from many sources.

As a consequence of these characteristics, we believe that our packet classifier is well suited to the construction of remote firewalls that service large nodes populations, where the nodes themselves dynamically update policy. The enforcement of policy on expensive link is a typical example of where such a firewall could be applied.

There are several areas where further investigation is warranted:

- How does the classifier perform for other typical scenarios?
- How does varying fundamental R*-tree construction parameters affect efficiency?
- Do other R-tree variants perform better or worse? What about more exotic SAMs?
- What are the consequences of defining more or fewer features? How effective is classification with statefully generated features?

References

[1] A. Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD Conference 1984: 47-57

[2] T. K. Sellis, N. Roussopoulos, C. Faloutsos: The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. VLDB 1987: 507-518

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger: The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD Conference 1990: 322-331

[4] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, Y. Theodoridis: R-trees Have Grown Everywhere. Unpublished Technical Report (2003). `http://www.rtreeportal.org/pubs/MNPT03.pdf`

[5] The OpenBSD Project. `http://www.openbsd.org/`

[6] Marios Hadjieleftheriou: Spatial Index Library. `http://www.cs.ucr.edu/~marioh/spatialindex/`