Virtual Switch Packet Classification

1 High Level Design Overview

1.1 Problem Statement

Packet classification is the problem of categorizing packets into *flows*, where all packets belonging to the same flow are processed by a predefined set of *rules*. Packet classification can be *N-dimensional*, i.e., can be on *N* different fields of packet header. For example, a two-dimensional packet classification may define a flow as "all packets destined to address A and port P", and choose to "block" all such flows.

Network devices solve packet classification by maintaining a database of flow-rule pairs. For every incoming packet, they search the database for a flow that matches the packet header, and then apply the matching rule. This procedure is referred to as *rule matching*. In practice, a packet might match multiple rules. To resolve ambiguity, rules are given priorities, and the “best” matching rule, i.e., the rule with the highest priority, is returned. This matching scheme is known as *priority-based matching* (as opposed to *longest-prefix matching* which is not our focus here).

The most intuitive approach to solve packet classification is to create a list of all flow-rule pairs, sort them by their priorities, and linearly search the list to find a match. This approach, already employed in the Virtual Filtering Platform (VFP), our virtual switch, does not scale well as the number of rules grows (beyond a few hundred rules). In fact, today allows tenants to define tens of thousands of rules (user-defined routes, ACLs, etc.), which well justifies the need for scalable and efficient packet classification.

Here, we introduce a novel approach to packet classification that replaces linear rule matching in our Virtual Switch, the Virtual Filtering Platform (VFP). Our approach builds a data structures for every condition type (e.g., CIDR IP, IP range, single ports, etc.). It then independently searches each data structure for a matching rule, and returns the best matching rule found from any of the data structures. We call each of these data structures a *classifier*. Each classifier is constructed for a specific condition type (e.g., trie for CIDR IP). Conversely, we say that a condition is *optimized* by a classifier. Our classification approach is efficient in both space and time; it does not replicate any rules (previous approaches did) and performs insertion/lookup in average constant time. Hence, it will significantly improve VFP’s SYN rate.

1.2 Summary of Key Design Decisions

- We add a flag to VFP groups to set the rule-matching approach: linear vs. condition-optimized.
- We use 12 classifiers to optimize conditions (6 for src conditions and 6 for dest conditions):
    - Hash table for src/dest single IPs (i.e., IP/mask where mask is 32 and 128 for IPv4 and IPv6 respectively)
    - Tries for src/dest CIDR IPs (i.e., IP/mask where mask is smaller than 32 and 128 for IPv4 and IPv6 respectively)
    - Interval tree for src/dest IP ranges
    - Hash Table for src/dest single ports
    - Interval tree for src/dest port ranges

    - Hash Table for GRE keys and VXLAN VNIs
- We do not impose any constraint on group of conditions in a condition-optimized group.
    - We can have one rule with CIDR destination IP and another rule with destination IP range.
    - We allow wildcards.
- Our rule matching is complete, i.e., we match all conditions (although optimization happens just on one condition).

2 Detailed Design

Given a set of rules $R=\{R_1, R_2, ...R_n\}$ and a set of classifiers $C=\{C_1, C_2, ..., C_m\}$, our goal is to partition R over C (i.e., insert every rule to one and only one classifier) such that "lookup + match" time for every rule is minimized. Then, we use classifiers to perform rule matching on every incoming SYN packet. In the remainder of this section, we answer the following questions:

1. What is a classifier?
2. How do we partition rules over classifiers?
3. How do we perform rule matching?
4. What is the performance of partitioning and matching?

2.1 Classifier

A classifier is a data structure that *optimizes* a condition for the purpose of rule matching. In other words, it makes "searching for a matching rule" and "matching against rule conditions" faster by a data structure that provides optimized operations for that condition type. Example classifiers we want to support include: hash table for source/destination single IPs, trie for source/destination CIDR IPs, interval tree for source/destination IP ranges, hash table for source/destination ports, interval tree for source/destination port ranges, and hash table for GRE key / VXLAN VNI. We also use linked list as a default classifier for any condition type that does not have a dedicated classifier.

Each classifier has a number of internal *nodes*, where each internal node points to a list of rules (or NULL). A classifier supports the following operations:

1. **Insertion**: This operation takes a rule and a rule condition and inserts the rule to a classifier (where rule condition matches the classifier's condition type) by creating proper nodes based on the given condition, and attaching the rule to (the rule list of) the node. For example, Rule R1 with source IP 1.1.1.1, dest IP 2.2.2.2/10, src port 10, dest port 20-30 and protocol wildcard can be inserted to the trie of destination IP (or the hash table of src port, ...). Note the list of rules attached to every node is always kept ordered by rule priorities. Also note that a rule might have multiple conditions for one condition type. For example, a rule might have 10 destination CIDR IPs and 20 source IP ranges. If we choose to insert this rule to the trie of destination IP, we insert the rule 10 times, once for each destination IP.
2. **Lookup**: Lookup takes a condition and returns a list of all matching rules, i.e., all rules that satisfy that condition. For example, one might query all rules that match source IP 2.2.2.2. Lookup will return NULL if no matching rule is found.
3. **Deletion**: Similar to insertion, deletion takes a rule and a rule condition, and deletes the rule from the classifier that represent the rule condition.

2.2 Partitioning Rules over Classifiers

Given only one classifier, partitioning is trivial: if a rule has a condition whose type matches the condition type of the classifier, insert the rule to the classifier; otherwise, insert the rule to the default linked list classifier. For multiple classifiers, we resort to heuristics to choose an optimal classifier for every rule. Our heuristic works as follows:

1. Initialize all available classifiers (except for the default linked list).
2. Insert every rule to every classifier. If a rule's condition type is wildcard for a classifier, skip inserting that rule to that classifier.
3. Calculate two costs for every rule in every classifier: a) lookup cost of the rule in the classifier, and b) the effect the rule on the other rules in the classifier. Define the "total cost" as a linear combination of the two costs (i.e., c*a+b where c is a constant).
4. Pick the classifier with the minimum total cost for every rule. If a rule is not inserted into any classifier, pick the default linked list classifier for that rule.
5. De-initialize and re-initialize all classifiers (this time initialize the default linked list classifier).
6. Insert every rule to its classifier (i.e., the classifier we picked in step 4).

At the end of this procedure, every rule is inserted into one and only one classifier.

Step 3 calculates two costs: lookup cost and rule effect. Lookup cost refers to the cost of finding a rule from a classifier, and matching a packet against the rule's conditions. Rule effect refers to the effect of a rule on other rules in a classifier. These two costs are further illustrated below.

2.2.1 Lookup Cost

Lookup cost can be divided into three sub-costs: 1) finding the classifier node that has the rule in its rule list, 2) traversing the rule list to get to the rule, and 3) matching against the rule's conditions. Note that only the first sub-cost is classifier dependent. The 3rd sub-cost accounts for the worst-case cost of matching since we do not make any assumption about the distribution of the packets. This is calculated by adding up all rule conditions except the conditions whose type matches the classifier condition type. Furthermore, the three sub-costs might not have the same weight, so ideally, we might want to assign weights to them.

Figure 1 shows an example of a trie built for 5 rules. We can look up rule R3 from this trie with cost of 6: 1) getting to the node that points to R2 and R3 with the cost of 1, 2) getting to R3 from the list head with the cost of 2, and 3) matching against R3's three source IPs with the worst-case cost of 3, so 1+2+3=6.

Rule	Destination IP	Source IP	Action
R1	110*	1*	Allow
R2	11*	110*	Allow
R3	11*	11*, 010*, 001*	Allow
R4	0*	0*	Block
R5	*	0*	Block

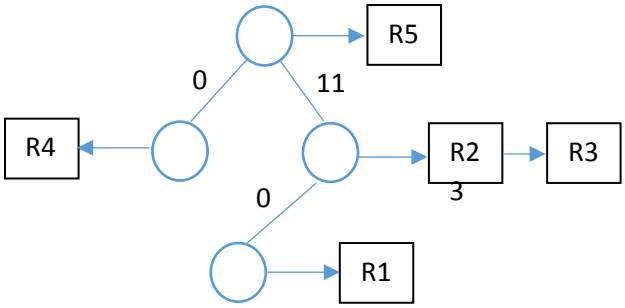

Figure 1. Trie for the rules in the table constructed based on destination IP.

Similarly, we can calculate the lookup cost in a hash table and an interval tree. In a hash table, the cost of getting to the right bucket is always 1, so we need to calculate the cost of traversing the rule list and matching against the rule. In an interval tree, the lookup cost is very similar to that of trie. Note that, in the presence of multiple classifiers, we might want to assign appropriate weights to the lookup cost from each classifier to make comparison across classifiers meaningful.

2.2.2 Rule Effect

The rule effect of R is defined as the effect of R on the other rules in the R's classifier with respect to the rule-matching procedure (see Section 2.3). This is calculated as the largest number of rules in the classifier that come "after" R. In a trie, this number is the largest number of rules in all the paths from R to all (reachable) leaves. Consider rule R5 in the Trie of Figure 1. There are two paths from R5 to the leaves, one that contains R4 and the other that contains R2, R3, and R1. So, the rule effect of R5 is 3. Similarly, the rule effect of R2 is 2, since R3 and R1 come after R2.

We can calculate the rule effect in a hash table and an interval tree in a similar fashion. In a hash table, we only need to account for the rules in one bucket. We calculate rule effect as the number of rules that come after one rule in the chain list of that bucket. In an interval tree, we calculate rule effect as the number of intervals that come after the interval and overlaps with it. In other words, this number is all the intervals whose start point is larger than the start point of the interval and smaller than the end point of the interval.

2.3 Rule matching

Give a partition of rules over classifiers, rule matching is performed as follows:

1. Search every classifier for a matching rule based on the relevant fields of the incoming packet.
2. Return the matching rule with the highest priority found from any of the classifiers.

If the above procedure does not find a matching rule, it will return NULL.

For example, let a flow be defined as a 2-tuple <src ip, dest ip>. Assume that we have 6 classifiers: hash table for src/dest single IPs, trie for src/dest CIDR IPs, and interval tree for src/dest IP ranges. Now imagine that a SYN packet <ip1, ip2> arrives. We extract source IP (i.e., ip1) from the packet and search our three classifiers on src IP (i.e., hash table, trie and interval tree) to find a rule that matches ip1. We perform similar procedure on our classifiers for dest IP (i.e., ip2). We compare the rules returned from the search procedures (in total 6 searches) and return the one with the highest priority (or NULL if no matching rule was found from any of the classifiers).

2.4 Performance

Our packet classification partitions rules over classifiers every time a new set of rules are added/deleted. It then invokes rule matching for every incoming (SYN) packet.

The overhead of partitioning is dominated by the cost of classifier insertion. We can divide insertion cost into two sub-costs: 1) getting to the right classifier node, and 2) inserting the rule to the node's rule list (while maintaining the list ordered by rule priorities). The first cost is constant for a hash table and a trie (note that the height of a trie is bounded), and is logarithmic for an interval tree. The second cost is linear in the number of rules in worst case, but constant in average. Overall, in average case, partitioning is linear in the number of rules (in fact, linear in the number of rule conditions).

The performance of rule matching is dominated by the cost of classifier lookup. Similar to the cost of classifier insertion, lookup is linear in worst case and constant in average case. VFP's original rule matching was linear in both worst and average cases. Improving the cost of rule matching from linear to constant for average case is a significant improvement.

3 Results

We generated 3 test cases with 1k, 10K, and 22K separate rules, where each rule randomizes src/dest IP subnets/ranges and src/dest single ports, and other fields left as wildcards. We also injected 20% wildcards independently for every field.

In all test cases, the matching rule is the last rule, i.e., the rule with the lowest priority. So, we're testing the worst-case scenario. Tests are run against a single core / single queue on a 2.2Ghz Sandy Bridge Xeon.

We ran a TCP SYN flood against VFP in these configurations and measured the rate at which SYNs wer processed and made it to the VM. The table below summarizes the results:

	1K random rules	10K random rules	22K random rules
Speed up with classifier search	6x	20x	35x