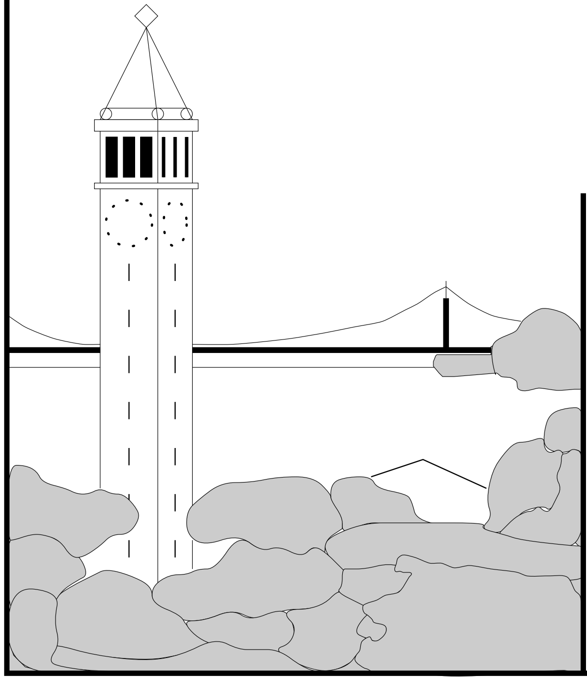
Efficient Multi-Match Packet Classification with TCAM

Fang Yu and Randy Katz
{fyu, randy}@eecs.berkeley.edu
CS Division, EECS Department, U.C.Berkeley



Report No. UCB/CSD-4-1316

March 2004

Computer Science Division (EECS)
University of California
Berkeley, California 94720

This technical report is supported by UC Micro grant number 03-041 and 02-032

Efficient Multi-Match Packet Classification with TCAM

Fang Yu, Randy H. Katz
EECS Department, UC Berkeley, Berkeley, CA 94720
E-mail: {fyu, randy}@eecs.berkeley.edu

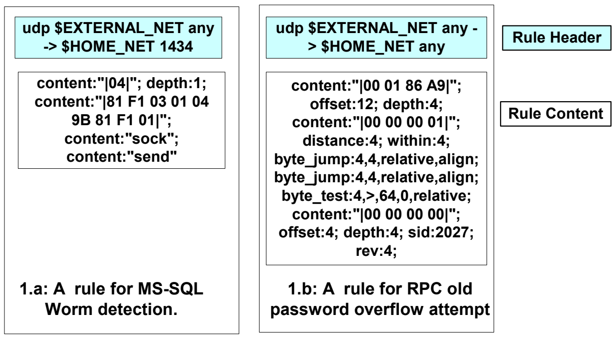
Abstract

Today's packet classification systems are designed to provide the highest priority matching result, e.g., the longest prefix match, even if a packet matches multiple classification rules. However, new network applications, such as intrusion detection systems, require information about all the matching results. We call this the multi-match classification problem. In several complex network applications, multi-match classification is usually the first step followed by other processing that is dependent on the classification results. Therefore, classification should be even faster than line rate. Pure software solutions cannot support such applications due to their slow speeds.

In this paper, we present a solution with Ternary Content Addressable Memory (TCAM), which produces multi-match classification results with only one TCAM lookup and one SRAM lookup per packet— about ten times fewer memory lookups than pure software solutions. In addition, we present a scheme to remove the negation format in rule sets, which can save up to 95% of TCAM space than the straight-forward solution. We show that using the pre-processing scheme presented in the paper, header processing for SNORT rule set can be done with one TCAM and one SRAM lookup using a 135KB TCAM.

1. Introduction

The emergence of new network applications demands multi-match classification, namely reporting all the matching results instead of only the highest priority match. One example of such applications is the network intrusion detection system, which monitors packets in a network and detects malicious intrusions or DOS attacks. In systems such as SNORT [1], there are thousands of rules. Figure 1.a is an example SNORT rule that detects a MS-SQL worm probe. Figure 1.b is a rule for detecting an RPC old password overflow attempt.



```
udp $EXTERNAL_NET any
-> $HOME_NET 1434

content:"|04|"; depth:1;
content:"|81 F1 03 01 04
9B 81 F1 01|";
content:"sock";
content:"send"
```

```
udp $EXTERNAL_NET any -
> $HOME_NET any

content:"|00 01 86 A9|";
offset:12; depth:4;
content:"|00 00 00 01|";
distance:4; within:4;
byte_jump:4,4,relative,align;
byte_jump:4,4,relative,align;
byte_test:4,>,64,0,relative;
content:"|00 00 00 00|";
offset:4; depth:4; sid:2027;
rev:4;
```

Rule Header
Rule Content
1.a: A rule for MS-SQL Worm detection.
1.b: A rule for RPC old password overflow attempt

Figure 1: SNORT rule examples.

Each rule has two components: a *rule header* and a *rule content*. The rule header is a classification rule that consists of five fixed fields: protocol, source IP, source port, destination IP, and destination port. The rule content is more complicated: it specifies intrusion patterns used for packet content scanning. Rule headers may have overlaps, so a packet may match multiple rule headers (e.g., both examples above). Multi-match classification is used to find all the rule headers that match a given packet so that we can check the corresponding rule contents one by one later.

Another application is programmable network elements (PNEs) [2, 3] proposed for implementing edge network functions. Typically, a packet traverses a number of network devices that perform different functions, e.g., firewall, HTTP load balancing, intrusion detection, NAT, etc. This can be highly inefficient because a packet has to traverse every device even if only a subset of them needs to operate on the packet contents. In addition, because each network device is separately built, common functions like classification are repeatedly applied. This wastes resources and induces extra delay. To address this problem, PNEs have been proposed to support multiple functions in one device. Multi-match classification is one important building block in PNEs: when a packet first enters a PNE, it is classified to identify the relevant functions. Then, only those selected functions will be applied, which saves resources and increases processing speed.

As we can see from the above two applications, multi-match classification is usually the first step in performing complex network system functions followed by processing that is dependent on the classification results. Applications that require only single-match classifications, however, tend to have further processing that is also simple (e.g., go to a specific port, or drop a packet, etc.). Therefore, to keep up the same line rate, multi-match classification may have to be much faster than single match to leave enough time for the complicated processing to follow without increasing latency too much.

Single-match packet classification is a well-studied problem. Given a packet, a classifier, which consists of a set of rules, reports the highest priority match. The single-match problem on multiple fields is a complex problem [4]. For n arbitrary non-overlapping regions in F dimensions, it is possible to achieve an optimized query time of $O(logn)$, with a complexity of $O(n^F)$ storage in the theoretical worst case [5, 6]. However, real-world rule sets are typically simpler than the theoretical worst case, and heuristic approaches [6, 7, 8] provide faster solutions, e.g., 20-30 memory accesses per packet in the "worst case".

The multi-match classification problem is more complex to implement than single-match classification since it needs all the matching results. Thus, some of the heuristic optimizations used for the single-match classification do not apply for multiple-match classification. Pure software solutions for multi-match classification are expected to take longer than that for single-match classification. However, as explained above, multi-match classification, because of the complex follow-up processing, is likely to have much tighter time requirements and hence pure software solutions are not likely to be sufficient.

In this paper, we present a scheme that provides a solution for the multi-match problem with two memory lookups: one using a Ternary Content Addressable Memory (TCAM), which is a type of memory that can do parallel search at high speed, and the other lookup using a standard Static Random Access Memory (SRAM). The remainder of the paper is organized as follows: we will begin by exploring some design choices and technical challenges in Section 2. Section 3 and 4 present our solution to the multi-match classification problem with TCAM. Finally we present simulation results in Section 5 and conclude in Section 6.

2. Motivation and Technical Challenges

A TCAM consists of many entries, the top entry of the TCAM has the smallest index and the bottom entry has the largest index. Each entry has several cells which can be used to store a string. TCAMs work as follows: given an input string, it compares the string against all entries in its memory in parallel, and reports the "first" entry that matches the input. The lookup time (5 *ns* or less) is deterministic for any input. Unlike a binary CAM, each cell in a TCAM can take one of three states: 0, 1, or 'do not care' (X). With 'do not care' states, a TCAM can support matching on variable prefix CIDR IP addresses and thus can be used in high-speed IP lookups [9, 10]. Also because it has 'do not care' states, one input may match multiple TCAM entries. In this paper, we assume the use of the widely-adopted *first-match TCAM*, which gives out the lowest index match of the input string if there are multiple matches as shown in Figure 2.

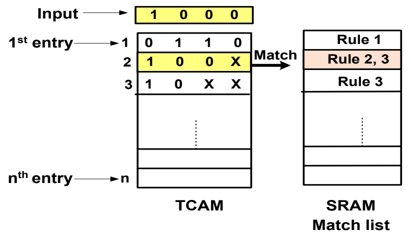


Figure 2: TCAM.

To solve the multi-match classification problem with TCAM, there are two challenges to be tackled: rule ordering and negation representation.

Challenge 1: Arrange rules in the TCAM compatible order

Rules can have different relationships such as subset, intersection, and superset. These relationships can cause problems for the matching results given a first-match TCAM. For example, suppose we have the following two rules:

(a) "Tcp $SQL_SERVER 1433 →$EXTERNAL_NET any"

(b) "Tcp Any Any → Any 139"

If we put rule (a) before rule (b) in the TACM, a packet matching both rules will report a match of (a) and never report (b), and vice versa. This is because rule (a) and (b) have an intersection relationship. Hence, we need to design an algorithm to add additional rules into the rule sets and order the rules in a specific way so that the above problem will not happen. We call such an ordering a "*TCAM compatible order*", which means: when a packet is compared with rules according to this order, we can retrieve all matching results solely based on the first matched rule. There should be no need to check the successive rules.

Challenge 2: Representing Negation with TCAM

Negation (!) operation is common in rule sets. For example, suppose we wish to find packets that are not destined to TCP port 80, we will use a rule *"tcp any any → any !80"*. The 16-bit binary form of 80 is 0000 0000 0101 0000. There is no direct way to map the negation into one TCAM entry. If we directly flip every bit over, 1111 1111 1010 1111 stands for 65375, which is only a subset of !80. To represent the whole range of !80, we need 16 TCAM entries. The basic idea is to flip one bit in one of the 16 binary positions and put 'do not care' to all the others. For example, the first entry would be 1xxx xxxx xxxx xxxx. The tenth entry would be xxxx xxxx x0xx xxxx, and so on.

In addition to port negation, some rules require subnets to be negated. For example, $EXTERNAL_NET frequently appears in rule sets, where $EXTERNAL_NET = !$HOME_NET. To represent this in TCAM directly, we need to flip every bit in the prefix of $HOME_NET and put 'do not care' to the other positions. Because IP subnet addresses are 32 bits, this costs up to 32 TCAM entries. Moreover, there could be several negations in one rule. For example, the rule "*tcp $EXTERNAL_NET any→ $ EXTERNAL_NET !80"* requires up to a total of 32*32*16=16384 TCAM entries for this single rule! This is obviously not an acceptable approach since TCAMs have a much smaller capacity than SRAMs (e.g., 1MB with current technology).

The next two sections describe our approach to addressing these two technical challenges.

3. Create Rules Sets in TCAM Compatible Order

To obtain multi-match results in one lookup with a first-match TCAM, we need to record intersections between rules. Studies in [4] show that the number of intersections between real-world rules is significantly smaller than the theoretical upper bound because each field has a limited number of values (e.g., all known port numbers) instead of unconstrained random values. So maintaining all the intersection rules is feasible. Indices of all those rules that used to generate the intersection are stored in a list. We call this a "Match List" and store the list in SRAM. Given a packet, we first perform a TCAM lookup and then use the matching index to retrieve all matching results with a secondary SRAM lookup as shown in Figure 2. The extended rules plus the original rules form an *extended rule set*. Throughout the remainder of this paper, a "rule" refers to a member of the extended rule set, unless otherwise specified as a member of the original rule set. The items in the match list are the indices of rules in the original rule set.

As defined in Section 2, the TCAM compatible order requires rules to be ordered so that the first match should record all the matching results in the match list. We first enumerate the relationships between any two different rules E_i and E_j, with match list M_i and M_j. There are four cases: exclusive, subset, superset, and intersection, each with following corresponding requirements:

1. Exclusive ($E_i \cap E_j = \phi$): then i and j can have any order.
2. Subset ($E_i \subseteq E_j$): then $i<j$ and $M_j \subseteq M_i$.
3. Superset ($E_j \subset E_i$): then $j<i$ and $M_i \subseteq M_j$.
4. Intersection ($E_i \cap E_j \neq \phi$): then there is a rule $E_l = (E_i \cap E_j)$ $(l<i, l<j)$, and $(M_i \cup M_j) \subseteq M_l$.

Case 1 is trivial: if E_i and E_j are disjoint, they can be in any order since every packet matching E_i never matches E_j. For Case 2 where E_i is a subset of E_j, every packet matching E_i will match E_j as well, so E_i should be put before E_j and match list M_i should include M_j. In this way, packets first matching E_i will not miss matching E_j. Similar operations are required for Case 3. Besides these three cases, partially overlapping rules lead to Case 4. In this case, we need a new rule E_l recording the intersection of these two rules ($E_i \cap E_j$) placed before both E_i and E_j with both match results included in its match list ($M_i \cup M_j) \subseteq M_l$). Note that the intersection of E_i and E_j may be further divided into smaller regions by other rules (e.g., E_k in Figure 3). In this case, all the smaller regions ($E_i \cap E_j$ and $E_i \cap E_j \cap E_k$) have to be presented before both E_i and E_j. This can actually be deduced by requirement (4).

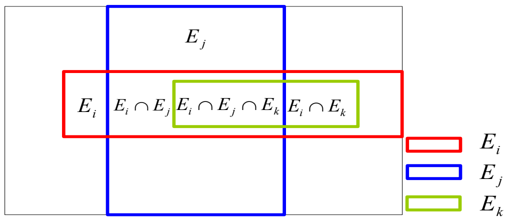


Figure 3: Example of intersection of three rules.

Case 1 to 4 covers all the possible relationships between any two rules. By applying the corresponding operations talked above, we can meet the requirements and get a TCAM compatible order.

```
Extend_rule_set(R){
        $E=\phi$;
        For all the rule $R_i$ in $R$
                $E$=Insert($R_i$, $E$);
        return $E$;
}
Insert(x, E){
    for all the rule $E_i$ in $E$ {
        Switch the relationship between $E_i$ and $x$:
            Case exclusive:
                continue;
            Case subset:
                $M_i = M_x \cup M_i$;
                continue;
            Case superset:
                $M_x = M_x \cup M_i$;
                add $x$ before $E_i$ ;
                return $E$;
            Case intersection:
                If ($E_i \cap x \notin E$ and $M_x \not\subset M_i$)
                    add $t = E_i \cap x$ before $E_i$ ;
                    $M_t = M_x \cup Mi$
    }
    add $x$ at the end of $E$ and return $E$;
}
```

Figure 4: Code for generating TCAM compatible order.

Figure 4 is the pseudo-code for creating a TCAM compatible order. The algorithm takes the original rule set $R=\{R_1, R_2, \ldots., R_n\}$ as the input. Each rule R_i is associated with a match list, which is index of itself ($\{i\}$). The algorithm will output an extended rule set E in the TCAM compatible order. The algorithm inserts one rule at a time into the extended rule set E, which is initially empty (the empty set obviously follows the requirements of TCAM compatible order). Next, we will show that after each insertion, E still meets the requirements. *Insert(x, E)* is the routine to insert rule x into E. It scans every rule E_i in E and checks the relationship between E_i and x. If they are exclusive, then we can bypass E_i. If E_i is a subset of x, we just add match list M_x to M_i and proceed to the next rule. If E_i is a superset of x, we should add x before E_i according to requirement (3) and ignore all the rules after E_i (see the proof in the appendix). Otherwise, if they intersect, then according to requirement (4), a new rule $E_i \cap x$ needs to be inserted before E_i if it is not presented before. The match list for the new rule is $M_x \cup M_i$. As you can see, we strictly follow the four requirements when adding every new rule, so the generated extended rule set E is in the TCAM compatible order. Due to space limitations, we do not go into the details of the deletion algorithm.

To better illustrate the algorithm, let's look at the following example in Table 1 which contains three rules. To generate extended rule set E, first we insert rule 1. Rule 2 does not intersect with rule 1 so it can be added directly. Now, we have rule 1 followed by rule 2. When inserting rule 3, we find that it intersects with both rule 1 and rule 2, so we add two intersection rules with match list {1, 3} and {2, 3} and put rule 3 at the bottom of the TCAM. The final extended rule set E is presented in Table 2.

	Original rule sets
1	Tcp \$SQL_SERVER 1433 $\rightarrow$ \$EXTERNAL_NET any
2	Tcp \$EXTERNAL_NET 119 $\rightarrow$ \$HOME_NET Any
3	Tcp Any Any $\rightarrow$ Any 139

Table 1: Example of original rule set with 3 rules.

Extended Rules	**Match List**
Tcp \$SQL_SERVER 1443 $\rightarrow$\$EXTERNAL_NET 139	1,3
Tcp \$SQL_SERVER 1433 $\rightarrow$\$EXTERNAL_NET any	1
Tcp \$EXTERNAL_NET 119 $\rightarrow$\$HOME_NET 139	2,3
Tcp \$EXTERNAL_NET 119 $\rightarrow$ \$HOME_NET any	2
Tcp any any $\rightarrow$ any 139	3

Table 2: Extended rule set in the TCAM compatible order.

4. Negation Removing

The scheme presented in Section 3 can be used to generate a set of rules in the TCAM compatible order. In this section, we describe how to insert them into TCAM. As explained before, each cell in the TCAM can take one of three states: 0, 1 or 'do not care'. Hence, each rule needs to be represented by these three states.

Usually, a rule contains IP addresses, port information, protocol type, etc. IP addresses in the CIDR form can be represented in the TCAM using the 'do not care' state. However, the port number may be selected from an arbitrary range. Liu[11] has proposed a scheme to efficiently solve port range problem, so we will not discuss this further in this paper. A more complicated problem for the TCAM is that some IP and port information is in a negation form. As explained in Section 2, each negation consumes many TCAM entries, so in this section, our goal is to remove negation from the rule set to save TCAM space.

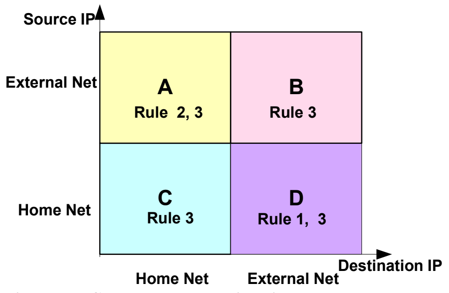


Figure 5: Source and destination IP addresses space.

Before we present our scheme, let us first look at the combinations of source and destination IP address spaces as shown in Figure 5. Use the rule set in Table 1 as an example, rule 3 applies to all 4 regions since it is "any" source to "any" destination; rule 1 applies to region D because we assume \$SQL_SERVER is in side \$HOME_NET; and rule 2 applies to region A. The regions that contain negation ($EXTERNAL_NET) are region A ($EXTERNAL_NET to $HOME_NET), B ($HOME_NET to $EXTERNAL_NET), and D ($EXTERNAL_NET to $EXTERNAL_NET).

Consider region A as an example: the rules in this region are in the form of *"* $EXTERNAL_NET * → $HOME_NET$^+$ *"*. Note that * means it could be any thing (e.g. "tcp" or "any" or a specific value). $HOME_NET$^+$ stands for $HOME_NET and any subset of it such as $SQL_SERVER. If we can extend rules in region A to region A and C, we can replace $EXTERNAL_NET with keyword "any" and now rules are in the format of *"* any * → $HOME_NET$^+$ *"*. However, after extending the region, we change the semantics of the rule and this may affect packets in region C. In another word, packet *"* $HOME_NET * → $HOME_NET$^+$ *"* will report a match of this rule as well.

This problem, however, is solvable because TCAM only reports the first matching result. With this property, we can first extract all the rules applying to region C and put those rules at the top of TCAM. Next, we add a separator rule between region C and region A*: "any $HOME_NET any → $HOME_NET any"* with an empty action list. In this way, all the packets in region C will be matched first and thus ignore all the rules afterwards. With this separator rule, we can now extend all the rules in region A to region A and C. Similarly, rules in region D can be extended to region C and D, rules in region B can be extended to region A, B, C, D. Therefore, we will put all the rules in the following order:

- Rules in region C: *"* $HOME_NET$^+$ * → $HOME_NET$^+$ *"*
- Separator rule 1: *"any $HOME_NET any → $HOME_NET any"*
- Rules in region D, specified in the form of region C and D: *"* $HOME_NET$^+$ * → any *"*
- Rules in region A, specified in the form of region A and C: *"* any * → $HOME_NET$^+$ *"*
- Separator rule 2: *"any $HOME_NET any → any any"*
- Separator rule 3: *"any any any → $HOME_NET any"*
- Rules applying to region B, specified in the form of region A, B, C and D: *"* any * → any *"*

Putting extended rule sets in the above order can be simply achieved by first adding all three separator rules to the beginning of the original rule set, then following the algorithm in section 2. If a rule applies to regions A, it will automatically intersect with the separator 1 and generate a rule in region C. If a rule applies region B, then it will intersect with all three separators and create three intersection rules. After that, we can replace all the $EXTERNAL_NET with keyword "any".

TCAM Index	TCAM entries	Match list
1	tcp $HOME_NET any → $HOME_NET 139	3
2	any $HOME_NET any → $HOME_NET any	
3	tcp $SQL_SERVER 1433 → any 139	3, 1
4	tcp $SQL_SERVER 1433 → any any	1
5	tcp any 119 → $HOME_NET 139	2,3
6	tcp any 119 → $HOME_NET any	2
7	tcp any any → any 139	3

Table 3: Extended rule set in a TCAM with no negation.

Table 3 shows the result of mapping the rule set of Table 1 into TACM. The first rule in region C is extracted from rule 3 that applies to all four regions. The second rule is a separator rule. With these two rules, we can replace the $EXTERNAL_NET in rules 3-6 with keyword "any". At the end, there is rule 7 which applies to all the regions. Separator rules 2 and 3 are omitted because no rule is in the form of $EXTERNAL_NET to $EXTERNAL_NET in the original rule set. In this example, by adding only two rules, we can completely remove the $EXTERNAL_NET. Compared to the solution in table 2 which needs up to 4*32 +1 = 129 TCAM entries, it is 94.5% of space saving!

The above example is a special case because there is only one type of negation ($EXTERNAL_NET) in one field. In a more general case, there can be more than one negation in each field. For example, there could be both !80 and !90 or !subnet1 and !subnet2 in the same field. Our scheme can be easily extended. If there are k unique negations in one field and their non-negation forms do not intersect (e.g., 80 and 90), then we need k separators of the non-negation form (80, 90) and they can be in any order. If they intersect, then we need up to $2^k -1$ separation rules for this field. For instance, suppose there are !subnet1 and !subnet2, there should be three separation rules applying to subnet1 $\cap$ subnet2, subnet2, and subnet1. k is usually a very small number because it is limited by a number of peered subnet. In general, if each field i needs k_i separators, then at most of $(\prod(k_i+1))-1$ separator rules should be added. In our previous example of removing $EXTERNAL_NET from source and destination IP addresses, $k_1 = k_1 = 1$, so we need a total of 2*2-1=3 separator rules.

5. Simulation results

To test the effectiveness of our algorithm, we use the SNORT [2] rule set. The SNORT rule set has undergone significant changes since 1999. We tested all the versions after 2.0 that are publicly available. Although each rule set has around 1700-2000 rules, many of the rules share a common rule header. As illustrated in Table 4, unique rule headers in each version are relatively stable. Note that we omitted the versions that share the same rule headers with the previous version.

Our task is to put these rule headers into TCAM as classification rules, and store the corresponding matching rule indices in the match list. Hence, given an incoming packet, with one TCAM lookup and another SRAM lookup, we can implement multi-match packet classification.

The second column in Table 5 records the size of extended rule set in the TCAM compatible order. It is roughly 10 times the size of the original rule set, which is well below the theoretical upper bound. This agrees with the findings in [4, 7, 8].

Version	Release Date	Rule Set Size	Rules added	Rules deleted
2.0.0	4/14/2003	240	-	-
2.0.1	7/22/2003	255	21	6
2.1.0	12/18/2003	257	3	1
2.1.1	2/25/2004	263	6	0

Table 4: SNORT rule headers statistics.

Version	# of rules in extended set	Single negation	Double negations	Triple negations
2.0.0	3,693	62.334%	0.975%	0
2.0.1	4,009	62.484%	1.422%	0.025%
2.1.0	4,015	62.540%	1.420%	0.025%
2.1.1	4,330	62.332%	1.363%	0.023%

Table 5: Statistics of extended rules set in TCAM compatible order.

Snort version	With Negation		Negation Removed		TCAM space saved
	Extended rule set size	TCAM entries needed	Extended rule set size	TCAM entries needed	
2.0.0	3,693	120,409	4,101	7,853	93.4%
2.0.1	4,009	145,208	4,411	8,124	94.4%
2.1.0	4,015	145,352	4,420	8,133	94.4%
2.1.1	4,330	151,923	4,797	8,649	94.3%

Table 6: Performance of negation removing scheme.

The number of negations in the extended rule set is significant. As shown in Table 5, on average 62.4% of the rules have one negation, 1.295% of the rules have two negations and there are even rules with three negations. In our simulation, we assume home network is a class C address that has a 24 bit prefix, so each $EXTERNAL_NET needs 24 TCAM entries. Negation of port, e.g., !80, !21:23 consumes 16 TCAM entries. Under this setting, a single negation takes up to 24 TCAM entries; a double negation consumes up to 24*24=576 TCAM entries; and a triple negation requires up to 24*24*16=9216 TCAM entries. Hence, if we directly put all the rules with negation into the TCAM, it takes up to 151,923 TCAM entries as shown in the third column of Table 6.

Our negation removing scheme in Section 3 can significantly save TCAM space. For the SNORT rule header set, we added 2*3*2*2-1 = 23 separation rules in front of the original rule set because there are four types of negations: $EXTERNAL_NET at source IP, $EXTERNAL_NET at destination IP, !21:23 and !80 at source port, and !80 at destination port. It only adds about 10% extra rules in the extended rule set (4th column of Table 6). However, with this 10% more rules, we can reduce the number of TCAM entries required by over 93%.

Note that the total number of required TCAM entries is larger than the extended rule set size. This is because some rules contain port ranges and consume extra TCAM entries. The range mapping approach in [11] is not used because this approach requires two additional memory lookups for key translations, and classification speed is our main concern. If a lower speed is acceptable, then we can also incorporate the range mapping technique and the total TCAM entries needed is just the size of extended rule set after removing negations.

Each rule is 104 bits (8 bits protocol id, 2 ports with 16 bits each, 2 IP addresses with 32 bits each), which can be rounded up to use a 128 bits entry TCAM. The total TCAM space needed for SNORT rule header set is 128*8649=135KB.

To study the effect of negation, we randomly vary the negation percentages in the original rule set. In the SNORT original rule header sets, 89.7% of rules contain single negation and 1.1% of the rules contain double negation. So, we first focus on the effect of single negation. Figure 6 shows the TCAM space needed both with and without our negation removing scheme. When the percentage of negation is very low, the two schemes perform closely. If we study closely, when the negation percentage is very small (<2%), putting negation directly is better than our scheme since we introduce extra separation rules that may intersect with other rules. However, as the percentage of negation is higher, the TCAM space needed for "with negation" case grows very fast. In contrast, the curve of our scheme remains flat and thus can save a huge number of TCAM space. For example, when 98% of the rules involve negation, our scheme can save 95.2% of the TCAM space compared to the "with negation" case. This is only for the single negation case. Due to space limitations, we do not present result for double negation cases. However, we can imagine that the saving would be even higher since each double negation rule requires more TCAM entries.

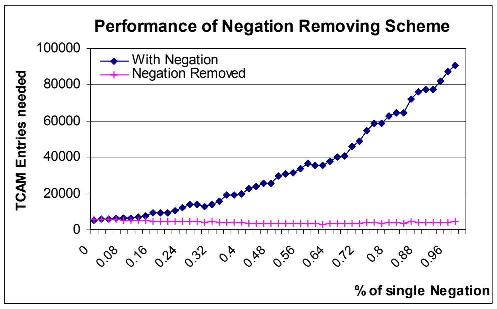


Figure 6: Negation removing scheme.

6. Conclusion

In this paper, we use a TCAM-based solution to solve the multi-match classification problem. The solution reports all the matching results with a single TCAM lookup and a SRAM lookup. In addition, we propose a scheme to remove negation in the rule sets which saves 93% to 95% of the TCAM space over the straightforward implementation. From our simulation results, the SNORT rule header set can easily fit into a small TCAM of size 135KB and is able to retrieve all matching results within two memory accesses. We believe a TCAM-based approach is viable, as TCAM is now becoming a common extension to network processors. Although TCAM is more expensive and has higher power consumption than standard memory such as DRAM and SRAM, the capability and speed it offer still make it an attractive approach for high speed networks.

References

[1] SNORT network intrusion detection system, www.snort.org,
[2] D. L. Tennenhouse and D. J. Wetherall, "Towards an Active Network Architecture," *Computer Communication Review*, Vol. 26, No. 2, April 1996
[3] G. Porter, M. Tsai, L. Yin, and R. Katz, "The OASIS Group at U.C. Berkeley: Research Summary and Future Directions," *http://oasis.cs.berkeley.edu/pubs/oasis_wp.doc*
[4] M. E. Kounavis, etc., "Directions in Packet Classification for Network Processors," *NP2 Workshop*, Feburary 2003
[5] M. H. Overmars and A. F. Stappen, "Range searching and point location among fat objects," *European Symposium on Algorithms,* 1994
[6] P. Gupta, N. McKeown "Packet classification using hierarchical intelligent cuttings," *in Hot Interconnects,* August 1999
[7] P. Gupta, N. McKeown "Packet classification on multiple fields," *in SIGCOMM,* August 1999.
[8] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet Classification Using Multidimensional Cutting," *in SIGCOMM*, August 2003
[9] E. Spitznagel, D. Taylor, and J. Turner, "Packet Classification Using Extended TCAMs," *ICNP*, November 2003
[10] P. Gupta, and N. McKeown, "Algorithms for Packet Classification," *IEEE Network,* March 2001
[11] H. Liu, "Reducing Routing Table Size Using Ternary-CAM", *Hot Interconnects,* August 2001

Appendix

Claim in section 3: If E_i is the first superset of x ($x \subset E_i$) in E, we can add x before E_i according to requirement (3) and bypass all the rules after E_i.

Proof: For any rule E_j after E_i, there could be four cases. We will study it one by one and show why we can bypass all of them.

First, we can bypass any rule E_j that is disjoint with x, according to requirement (1).

Second, it is impossible that $E_j \subset x$. If so, $E_j \subset x \subset E_i$, which contradicts with requirement (2).

Third, If $x \subset E_j$, E_j must also be a superset of E_i. Otherwise, the intersection of E_j and E_i must be a superset of x as well and it must be presented before E_i, according to requirement (4). This contradicts with the assumption that E_i is the first superset of x in E. Therefore, $E_i \subset E_j$ and we have $M_j \subset M_i$ according to requirement (2). In this case, we don't need to process E_j since we can extract all the information from M_i.

Fourth case, if E_j intersects with x and suppose $z = E_j \cap x$, then z must have appeared before E_i. This is because E_j must intersect with E_i as well since E_i is a superset of x. Let $E_k = E_i \cap E_j$, according to requirement (4), $k < i$. In addition, $z = E_j \cap x = E_j \cap x \cap E_i = E_k \cap x$, because $x \subset E_i$. Therefore, we must have generated z when processing E_k which is before E_i. This meets the requirement (4), so we can bypass E_j.

Hence, all the rules after E_i are either exclusive to x, or their intersections have already been included, so we can skip all those rules.