# TCAM REDUCTION TECHNIQUES FOR ALL-MATCH CLASSIFIERS

By

Nicholas Jon Wender

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Computer Science

2012

**Abstract**

**TCAM REDUCTION TECHNIQUES FOR ALL-MATCH CLASSIFIERS**

**By**

**Nicholas Jon Wender**

Network intrusion detection systems require all-match packet classification, where all rules matching a packet are reported by the system. The problem of efficiently reporting all matching rules is known as the *all-match optimization problem*. One solution is to convert an all-match classifier into a first-match classifier (in which only the first classifier rule that matches a packet is reported), and use ternary content addressable memory (TCAM) for packet classification.

In this thesis, we evaluate two classifier minimization approaches. First, we consider the use of all-match classifier-specific optimization algorithms. Second, we use state-of-the-art first-match classifier optimization algorithms in conjunction with all-match algorithms. Our results indicate the appropriate approach is related to the number of TCAM chips available for classification. When using one TCAM chip, we attain 70.85% TCAM space savings using first-match classifier optimization algorithms instead of all-match classifier optimization algorithms. When using multiple TCAM chips, we found that it is best to use all-match specific optimization algorithms.

# Contents

# List of Tables

# List of Figures

# KEY TO SYMBOLS AND ABBREVIATIONS

| Symbol | Description | | Symbol | Description |
|---|---|---|---|---|
| $b$ | Number of bits | | $\mathbb{N}_i$ | Set of negated fields for $\mathbb{F}_i$ |
| $d$ | Number of dimensions | | $n$ | Number of literals/rules |
| $\mathbb{C}$ | Classifier | | $m$ | Number of clauses/intersections |
| $r_i$ | Rule $i$ of a classifier | | $L$ | Set of literal pairs |
| $d(r)$ | Decision of rule $r$ | | $x_i$ | Positive literal for rule $i$ |
| $\mathbb{F}_i$ | Field $i$ | | $C$ | Clause |
| $r[i]$ | Field $i$ of rule $r$ | | $R$ | Subset of $\mathbb{F}$ containing non-intersection rules |
| $D(\mathbb{F}_i)$ | Domain of field $i$ | | $\mathbb{I}$ | Subset of $\mathbb{F}$ containing intersection rules |
| $p$ | Packet | | $I$ | Intersection rule |
| $p[i]$ | Field $i$ of packet $p$ | | $C_{Ii}$ | Clause $i$ for intersection $I$ |
| $p \subseteq r$ | Packet $p$ matches rule $r$ | | $f_i$ | FDD $f$ for field $i$ |
| $\mathbb{C}(p)$ | Classification of $p$ using $\mathbb{C}$ | | $DS$ | Decision set of a classifier for an FDD |
| $\mathbb{AM}$ | All-match classifier | | $v$ | FDD node |
| $\mathbb{FM}$ | First-match classifier | | $e$ | FDD edge |
| $m(r_i)$ | Matchlist of $r_i$ | | $F(v)$ | Label of FDD node $v$ |
| $\mathbb{R}$ | Region | | $I(e)$ | Set of labels for FDD edge $e : u \to v$ |
| $D(\mathbb{R})$ | Domain of region $\mathbb{R}$ | | $L_i$ | Landmark range for edge $i$ |
| $P$ | Set of packets | | $\mathbb{T}_i$ | Transformer for field $i$ |
| $s$ | Separator rule | | $S_i$ | Range of rule from field $i$ |

# Chapter 1: Introduction

Network intrusion detection/prevention systems (NIDS/NIPS) detect and prevent malicious attacks on networks. Most NIDS use a set of rules, called a classifier, that define how to handle network traffic. Given a packet, the system searches the classifier for rules that match the packet. These rules specify how the matched packet should be processed.

The Snort intrusion detection system is a NIDS with a publicly available set of rules [1]. The Snort rules consist of two parts: a rule header and a set of rule options (signature). The rule header is a filter and an action. We use the filter to determine if a given packet matches the rule header. For a given packet and rule, if the packet header matches the rule header's filter and the packet payload matches the rule options, then we apply the rule's action to the packet.

Our focus in this thesis is on the first step in the above classification procedure. We are only concerned with matching packet headers to rule headers; we do not worry about matching the packet payload to the rule options. By design, the Snort rules are in *all-match* (also called *multi-match*) semantics. This means that a packet can match multiple rule headers. We define the *all-match* or *multi-match optimization problem* as the problem of finding all rules in a given classifier that match a given packet.

One solution to the all-match optimization problem is to convert a given all-match classifier to a *first-match classifier*, which have been extensively studied. A first-match classifier is a classifier in which we only return the first (or highest priority) rule matching a given packet. The conversion process takes our all-match classifier and creates a semantically equivalent first-match classifier such that we can find all matching rules for any packet.

Given a first-match classifier, our challenge is the *first-match optimization problem*, in which we want to minimize the size of the classifier. There are many solutions to the first-

match optimization problem, including the very popular solution of using ternary content addressable memory (TCAM) to store the classifier. TCAM chips are widely used because they provide a constant time lookup for all packets. However, TCAM suffers from several drawbacks: high cost, high power consumption, and limited size.

In the end, we are left with two approaches when converting an all-match classifier into a first-match classifier: (1) We can use all-match optimization techniques like Negation Removal and the Set Splitting Algorithm (SSA) to reduce the size of the resulting first-match classifier. (2) We can simply convert the all-match classifier into a first-match classifier and then apply first-match optimization algorithms to reduce the size of the classifier.

We begin with a short discussion of ternary content addressable memory and follow with an overview of the Snort network intrusion detection system.

## 1.1 Ternary Content Addressable Memory

Hardware based packet classification using ternary content addressable memory (TCAM) has become the industry standard for high performance firewalls and Internet routers [6]. Each TCAM entry holds one rule, and each cell of a TCAM entry can take one of three states: 0, 1, or * — a special "don't care" state that allows a bit to be either a 0 or 1. Given a packet as a search key, the underlying TCAM circuitry search all entries in parallel and return the index of the first-matching (or highest priority) entry. For example, the TCAM shown in figure 1.1 searches all entries in parallel for the key 10100. We can see from the highlighted rows that the search key matches two entries; however, only the index of the first matching entry will be returned.

While providing a deterministic lookup time for a given packet [18], TCAM suffers from

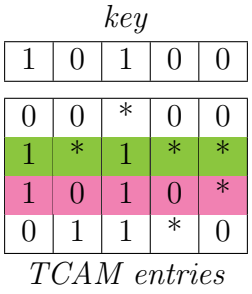


Figure 1.1: Using a TCAM to search for a given key in parallel. For interpretation of the references to color in this and all other figures, the reader is referred to the electronic version of this thesis

several well known disadvantages. TCAM chips are much more expensive to manufacture, have smaller capacities, and consume significantly more power than a comparable amount SRAM [18]. The complex circuitry that allows TCAM to perform a parallel search is not only costly, but also uses additional board space and consumes more power. Similarly while TCAM chips of up to 72 Mbit have been produced, smaller TCAM chips are more widely used due to the high cost.

Another issue faced by TCAM is the extensively studied range expansion problem. While prefixes (discussed in detail in a section 4.3) can encode the IP address and protocol fields for classifiers, ranges given for the port fields usually cannot map directly into one TCAM entry. Thus TCAM chips need to use multiple entries to encode a single range. A $b$-bit field needs to be split into at most $2(b-1)$ separate entries [18]. Therefore, each 16-bit port field can require up to 30 TCAM entries in the worst case. In addition, if a rule has a range for both port fields, it can require up to $30x30 = 900$ TCAM entries. Many strategies have been developed to deal with the range expansion problem [2, 3, 8, 12, 17, 14, 16].

A third challenge TCAM chips have is the presence of negated rules. A negated rule is a rule that has a field that is negated, for example `TCP 74.12.200.100 43` $\rightarrow$ `10.1.0.1 !43`. Consider the negated destination port `!43`. In binary, `43 = 0000 0000 0010 1011`

| 1*** **** **** **** |
|---|
| *1** **** **** **** |
| **1* **** **** **** |
| ***1 **** **** **** |
| **** 1*** **** **** |
| **** *1** **** **** |
| **** **1* **** **** |
| **** ***1 **** **** |
| **** **** 1*** **** |
| **** **** *1** **** |
| **** **** **0* **** |
| **** **** ***1 **** |
| **** **** **** 0*** |
| **** **** **** *1** |
| **** **** **** **0* |
| **** **** **** ***0 |

Figure 1.2: Naïve negation removal of the port `!43` by flipping bits.

and the naïve approach (see figure 1.2) for putting this negated field in TCAM would create sixteen entries, flipping one bit in each and replacing the rest with a "don't care" value. Thankfully there are more intelligent solutions to the problem of negated fields, which we will discuss this topic at length in section 4.2.1.

## 1.2 Snort Rules

Snort is an open source rule-based network intrusion detection and prevention system that can perform real-time packet analysis including content searching and protocol analysis. While Snort can perform simpler operations like packet logging and traffic sniffing, we only concern ourselves with Snort's network intrusion detection mode [1].

As briefly mentioned above, the Snort system is a rule-based system that relies on signature detection for packet classification. To Snort, both the packet header and the packet payload (the data contained in the packet) are important elements. Snort inspects the packet

payload for any packet whose header matches a Snort rule header. For this, Snort searches the packet's data according to the options specified in the corresponding rule. There are two aspects to this system that are worth noting. First, an arbitrary packet can match many distinct Snort rule headers. Second, each Snort rule header may correspond to several signatures.

| | | |
|---|---|---|
| Rule 1 | Filter: | `TCP $EXTERNAL_NET any → $SQL_SERVERS 3306` |
| | Options: | `content:"\|0A 00 00 01 85 04 00 00 80\|root\|00\|";` |
| | Action: | `alert msg:"MYSQL root login attempt"` |
| Rule 2 | Filter: | `TCP $EXTERNAL_NET any → $SQL_SERVERS 3306` |
| | Options: | `content:"\|0F 00 00 00 03\|show databases";` |
| | Action: | `alert msg:"MYSQL show databases attempt"` |
| Rule 3 | Filter: | `TCP $EXTERNAL_NET 21 → $HOME_NET any` |
| | Options: | `isdataat:1023;` `pcre:"/\d{3}\s+[^\n]{1019}/smi";` |
| | Action: | `alert msg:"FTP Microsoft Internet Explorer FTP Response Parsing Memory Corruption"` |

Figure 1.3: Example Snort rules with Filter, options, and actions.

For example, consider the three Snort rules shown in figure 1.3.

Any packet over the TCP protocol originating from any port outside the network and destined for an SQL server on port 3306 (any packet matching `TCP $EXTERNAL_NET any → $SQL_SERVERS 3306`) will match the header of the first two rules; thus, such packet payloads will be searched for the signatures in both rules. Any incoming packet (any packet going from `$EXTERNAL_NET` to `$HOME_NET`) on port 21 will match the header of the third rule; thus such packet payloads will be searched for the third rule's signature.

Furthermore, given that `$SQL_SERVERS` is defined as a subset of `$HOME_NET`, suppose that an incoming packet matched the following fields: `TCP $EXTERNAL_NET 21 → $SQL_SERVERS 3306`. This packet, would then match all three headers; thus, its payload would have to be checked for all three signatures. Admittedly, this is a contrived example because it is not

likely that a MySQL server will be accepting FTP connections on port 3306. However, the indicated rules illustrate the major point well: a packet may match many headers and the packet's payload needs to be inspected accordingly.

## 1.3 Problem Overview

From the preceding discussions of TCAM and the Snort system, we summarize the main challenge we address in this thesis. We wish to construct the smallest possible TCAM classifier that returns all rules that match a given packet header. We will state this problem formally in chapter 3.

## 1.4 Contributions

To solve the all-match optimization problem using TCAM chips, we implement previous solutions (all-match to first-match conversion with negation removal [22] and SSA [23]) in conjunction with additional first-match optimizations: TCAM Razor [11], Topological Transformation [14], and TCAM SPliT [15]. The objective of this thesis is to evaluate two approaches to the all-match optimization problem. (1) Evaluate the performance of optimization algorithms designed specifically for all-match classifiers. (2) Convert an all-match classifier to a first-match classifier and evaluate general purpose first-match classifier optimization algorithms.

We evaluate the performance and efficiency of various combinations of these algorithms by testing them on three different Snort rules. When using only one TCAM, our results show that TCAM Razor outperforms Negation Removal with TCAM space savings of 70.85%. When using multiple TCAM chips, it is best to use SSA to reduce the all-match classifier.

Stated another way, with one TCAM we should use general purpose first-match classifier optimization algorithms, and with multiple TCAM chips, we should use all-match specific optimization algorithms.

The remainder of this thesis proceeds as follows. In chapter 2 we discuss related work for the all-match optimization problem and first-match classifier optimization. In chapter 3 we define our notation and present a formal statement of the all-match optimization problem. Chapter 4 outlines technical details of our experiments. Chapter 5 presents our simulation results, and chapter 6 offers conclusions.

# Chapter 2: Related Work

In this chapter we review the relevant prior work in both all-match and first-match optimization. First we review the prior art of all-match optimization.

## 2.1 All-Match Optimization

While there is a surprisingly small volume of work in the area of all-match packet classification, the related work can roughly be split into two categories: (1) FPGA-based solutions and (2) TCAM-based solutions.

FPGA-based solutions use field programmable gate arrays (FPGAs) to implement all-match classification methods. A review of the work on all-match classification using FPGAs yields a fair body of research. However, we prefer TCAM-based solutions because many network processors already use TCAM chips for first-match classification.

As discussed earlier, TCAM is the industry standard for high performance devices (such as firewalls and routers) because TCAM chips provide line rate (or near line rate) classification that software solutions cannot match [20].

The current state of the art in TCAM-based all-match packet classification is the work by Yu et al. presented in [22] and [23]. In addition TCAM-based solutions have been proposed in [9] and [21].

The solution presented in [9] is intended to scale to larger databases as it requires no additional TCAM entries. However, this approach requires significantly more TCAM lookups to find all matching results. The work of [21] encodes classifier rules and performs both multiple one-dimensional TCAM lookups and hash table accesses.

The geometric approach in [22] transforms an all-match classifier into a first-match clas-

sifier which can be stored in TCAM such that with one additional SRAM lookup, we can obtain all matching rules for a given packet. However, this solution can drastically increase the number of rules that need to be stored. Yu et al. extend the work of [22] with [23], in which the resulting first-match classifier is split into two or four sets to reduce the number of additional rules that need to be stored. This approach requires only a handful of extra TCAM entries at the cost of requiring two or four lookups to classify each packet. We build upon these two all-match optimization techniques in the rest of this thesis.

## 2.2 First-Match Optimization

First-match packet classification has been extensively studied in the literature (See [6, 20] for a general overview). In the broadest sense, packet classification can be split into two categories, algorithmic and architectural. The industry standard TCAM-based architectural approach resulted from the inability of algorithmic solutions to provide fast enough performance. However due to the limitations of TCAM, combinations of algorithmic and architectural solutions are becoming more common [20]. Nevertheless, this thesis is not intended to be a comprehensive survey of packet classification algorithms.

For this thesis, we apply several state-of-the-art TCAM-based first-match classifier optimization algorithms to reduce the size of a first-match classifier generated from an all-match classifier. We focus on the following three approaches: TCAM Razor [11], Topological Transformation [14], and TCAM SPliT [15]. We describe these methods in more detail in section 4.3.

# Chapter 3: Definitions and Problem Statement

As we mention in the introduction, firewalls and Internet routers classify packets based on sets of rules called classifiers. In this chapter we formally define intervals, fields, rules, packets, and classifiers (both all-match and first-match classifiers). Then we formally state the problem we study in this thesis. These definitions provide a formal basis to discuss the all-match optimization problem throughout the rest of this thesis.

## 3.1 Definitions

We define a *classifier* as a set of rules $\mathbb{C}$. Each *rule* $r \in \mathbb{C}$ is in turn defined as a set of *fields* that is associated with an *action* or *decision*. The decision for $r$ defines the result imposed upon packets matching $r$ by $\mathbb{C}$. Let $r_i$ be the $i^{th}$ rule of classifier $\mathbb{C}$. Then we have that $d(r_i)$ is the action of $\mathbb{C}$ on $r_i$.

In general a rule has $d$ fields, denoted as $F_i$ for $1 \leq i \leq d$, and each rule is said to be a $d$-tuple, $r = (r_i, ..., r_d)$ over fields $F_1, ..., F_d$. We say that a $b$-bit field $F_i$ is defined over the domain $D(F_i) = [0, 2^b - 1]$. For convenience, we denote the $i^{th}$ field of rule $r$ with the notation $r[i]$. Typically $r[i] = [j, k]$ is an *interval*[1] such that $[j, k] \in D(F_i)$. Finally, we say that a classifier containing rules with $d$ fields is a $d$-dimensional classifier.

In practice, five commonly used fields are an 8-bit protocol, 32-bit source and destination IPv4 addresses, and 16-bit source and destination ports. We note that as IPv6 becomes more common, the 32-bit IPv4 address fields will become 128-bit fields. This will also increase the difficulty of packet classification as it will greatly increase the width of each rule.

[1] When working with the Snort rules, we relax our notion of an interval such that the field of a rule can be a *set* of intervals such that each interval in the set if in the domain of the field.

For classification, we say that a packet *matches* a rule if each field of the packet matches the corresponding field of the rule. More formally, a packet $p$ is a $d$-tuple $(p_1, ..., p_d)$ over fields $\mathbb{F}_1, ..., \mathbb{F}_d$. We overload the notation we use with rules for packets and say that field $\mathbb{F}_i$ of packet $p$ is $p[i]$ for $1 \leq i \leq d$. The formal difference between a rule $r$ and a packet $p$ is that $p[i] = v$ for $1 \leq i \leq d$ where $v$ is some discrete value in $D(F_i)$. Packet $p$ is said to match rule $r$ (denoted as $p \subseteq r$) if $p[i] = v \in [j, k] = r[i]$ for $1 \leq i \leq d$. If $p \subseteq r$, we set the decision of the packet, $d(p) = d(r)$. Let $\mathbb{C}(p)$ be the procedure through which $\mathbb{C}$ classifies $p$. Further let $\mathbb{C}(p) = \{r_1, ..., r_k\}$ be the set of $k$ rules that $p$ matches sorted in descending order by priority. Formally, we have $p \subseteq r_j \ \forall \ r_j \in \mathbb{C}(p)$.

Now we define the types of classifiers we consider in this thesis. First let us note that for any classifier $\mathbb{C}$ we may have some packet $p$ such that $p \subseteq r_i$ and $p \subseteq r_j$ for rules $\{r_i, r_j\} \in \mathbb{C}$ where $r_i \neq r_j$. That is, in any classifier there may exist a packet that matches more than one rule of the classifier. Let $r_i$ and $r_j$ be two distinct rules matched by $p$. Without loss of generality, let us say that $r_i$ has a *higher priority* than $r_j$ — which in practice implies that $r_i$ appears before $r_j$ in $\mathbb{C}$. We say that $\mathbb{C}$ is a *first-match classifier* if for any packet $p$, $\mathbb{C}(p)$ is exactly the set $\{r_i\}$, the highest priority rule matching $p$. Then, $|\mathbb{C}(p)| = 1$ for all packets $p$. On the other hand, we say that $\mathbb{C}$ is an *all-match classifier* if $\mathbb{C}(p)$ is the set $\{r | p \subseteq r\}$. Typically, $|\mathbb{C}(p)| > 1$ for most packets.

## 3.2 Problem Statement

Commonly referred to as the *multi-match* or *all-match optimization problem*, our goal is to transform an all-match classifier into the smallest equivalent first-match classifier. More specifically, given a classifier $\mathbb{C}$, we must generate a first-match classifier $\mathbb{C}'$ such that for any

packet $p$, we obtain all matching rules of $p$ in $\mathbb{C}$ by computing $\mathbb{C}'(p)$. Since $\mathbb{C}$ is an all-match classifier, $\mathbb{C}(p) = \{r_i, ..., r_k\}$. Likewise, because $\mathbb{C}'$ is a first-match classifier, $\mathbb{C}'(p) = \{r'_i\}$. Therefore our task is to encode $d(r'_i)$ so that we can compute $\{r_i, ..., r_k\}$ given $d(r'_i)$.

# Chapter 4: Algorithms

In this chapter, we provide an overview of all algorithms we evaluate. We begin with the all-match to first-match conversion algorithm from [22]. This algorithm is the basis of our work, as it allows us to create a first-match classifier $\mathbb{FM}$ from an all-match classifier $\mathbb{AM}$.

Following this discussion, we present two all-match specific optimizations: Negation Removal and the Set Splitting Algorithm (SSA). Negation removal allows us to reduce the number of rules required to represent negated fields in TCAM, while SSA enables us to reduce the number of intersection rules we need to store in TCAM.

After the sections on all-match algorithms, we present the first-match algorithms we evaluate. We begin by presenting the concepts of prefixes and firewall decision diagrams. Then we review three first-match optimization algorithms: TCAM Razor, Topological Transformation, and TCAM SPliT.

## 4.1 All-Match to First-Match Conversion

The initial step in solving the all-match optimization problem is to convert the given all-match classifier, $\mathbb{AM}$, into an equivalent first-match classifier, $\mathbb{FM}$. We follow the algorithm presented by Yu et al [22] to transform $\mathbb{AM}$ into $\mathbb{FM}$.

Before we discuss the relationships defined for rules, we explain the concept of a rule's *matchlist* and discuss how we use this concept with a first-match classifier. With each rule $r_i \in \mathbb{FM}$ we associate a matchlist, denoted $m(r_i)$, that enumerates the rules, $r_j \in \mathbb{AM}$, that packets matching $r_i$ in $\mathbb{FM}$ will match in $\mathbb{AM}$. More precisely, the matchlist $m(r_i)$ for rule $r_i \in \mathbb{FM}$ defines a subset of rules from $\mathbb{AM}$ such that for any packet $p$: if $p \subseteq r_i$, then $p \subseteq r_j$ $\forall\ r_j \in m(r_i)$.

The all-match to first-match algorithm works as follows: (1) Create $\mathbb{FM}$ from $\mathbb{AM}$ and store $\mathbb{FM}$ in TCAM. (2) Compute a matchlist for every $r_i \in \mathbb{FM}$ and store all matchlists in SRAM (3) For $r_i \in \mathbb{FM}$, assign $d(r_i) = k$ where $k$ is the SRAM index of $m(r_i)$.

Then to compute $\mathbb{AM}(p)$ for any packet, we perform a TCAM lookup to get $\mathbb{FM}(p) = d(p)$. Then we perform an SRAM lookup of $d(p)$ to obtain the matchlist for $p$.

Now we detail the possible relationships rules can have. These relationships are the foundation that we will use to compute $\mathbb{FM}$ from $\mathbb{AM}$. Between any pair of rules $r_i$ and $r_j$ in $\mathbb{AM}$, there exists a relationship (defined over all packets) that satisfies one of the following:

1. $r_i \cap r_j = \varnothing$. We say that $r_i$ and $r_j$ are *exclusive*. If two rules are exclusive, then no packet will match both rules and we can place the rules in any order relative to each other in $\mathbb{FM}$.

2. $r_i \subseteq r_j$. If $r_i$ is a *subset* of $r_j$, then we have that any packet that matches $r_i$ will also match $r_j$. Thus, we should place $r_i$ before $r_j$ in $\mathbb{FM}$ and add $j$ to $m(r_i)$.

3. $r_i \cap r_j \neq \varnothing$ (such that $r_i \not\subseteq r_j$ and $r_j \not\subseteq r_i$). If $r_i$ and $r_j$ have a non-empty intersection, then we need to create rule $r_k = r_i \cap r_j$, and place $r_k$ before $r_i$ and $r_j$ in $\mathbb{FM}$, and add $m(r_i)$ and $m(r_j)$ to $m(r_k)$.

Let us explicitly state how we use the preceding relationships to convert $\mathbb{AM}$ to $\mathbb{FM}$. Initially $\mathbb{FM}$ is an empty classifier and we sequentially insert each rule $r \in \mathbb{AM}$ into $\mathbb{FM}$ leveraging these relationships. Let $r_i$ be the current rule from $\mathbb{AM}$ we are inserting. First we make $m(r_i) = (i)$. Then for and each existing rule of $\mathbb{FM}$, $r_e$, we take one of the following actions based on the relationship between $r_i$ and $r_e$:

1. If $r_i$ and $r_e$ are exclusive, we take no action and proceed to the next rule in $\mathbb{FM}$.

2. If $r_i$ is a subset of $r_e$, we insert $r_i$ before $r_e$, append $m(r_e)$ to $m(r_i)$, and proceed to the next $r_i$ in $\mathbb{AM}$[1].

3. If $r_e$ is a subset of $r_i$, we append $m(r_i)$ to $m(r_e)$ and proceed to the next $r_e$ in $\mathbb{FM}$.

4. If $r_i$ and $r_e$ have a non-empty intersection, we create a new rule $r_k = r_i \cap r_e$, insert $r_k$ before $r_e$, append $m(r_e)$ to $m(r_k)$, append $m(r_i)$ to $m(r_k)$, and proceed to the next $r_e$ in $\mathbb{FM}$.

The conversion process from in $\mathbb{AM}$ to in $\mathbb{FM}$ will produce many additional rules from step 4. We refer to these rules as *intersection rules*.

After transforming $\mathbb{AM}$ to $\mathbb{FM}$, we must clean up $\mathbb{FM}$ to prepare $\mathbb{FM}$ for further reduction. First, the rules in $\mathbb{FM}$ may contain fields with discontinuous ranges. However, the first-match algorithms we use require that each field of a rule is defined over a continuous range. Thus, we split each rule that has discontinuous field ranges into multiple rules. Figure 4.1 illustrates the rule splitting process.

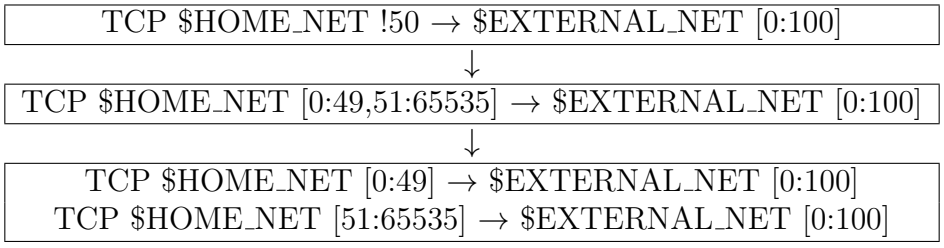


Figure 4.1: Splitting discontinuous rule ranges.

Second, the rules in $\mathbb{FM}$ have a matchlist as the decision, but the first-match classifier optimization algorithms we apply require that each rule has a single integer action. Thus, we map each distinct matchlist to a distinct integer. This mapping must be maintained so that after optimizing $\mathbb{FM}$, we can perform the SRAM lookup to find the corresponding $\mathbb{AM}$.

[1] Yu et al. [22] present a proof of why we can skip the rest of the existing rules.

| No. | Rule |
|---|---|
| 1 | TCP $HOME_NET !50 → $HOME_NET [0:100] |
| 2 | TCP $HOME_NET 50 → $HOME_NET 75 |
| 3 | TCP $HOME_NET any → $HOME_NET 150 |

↓

| | |
|---|---|
| 1 | TCP $HOME_NET any → $HOME_NET [0:100] |
| 2 | TCP $HOME_NET 50 → $HOME_NET 75 |
| 3 | TCP $HOME_NET any → $HOME_NET 150 |

Figure 4.2: Simple application of negation removal losing classifier semantics.

## 4.2 All-Match Optimization

When we transform $\mathbb{AM}$ into $\mathbb{FM}$, we can apply existing all-match optimization schemes designed specifically to minimize the resulting first-match classifier. As we discussed previously, representing negated fields in TCAM leads to a large growth in the number of TCAM entries. To mitigate this problem, we use a negation removal algorithm during the conversion process [22]. In addition, the first-match conversion process generates many additional intersection rules. We can limit the number of intersection rules by applying the SSA algorithm [23].

### 4.2.1 Negation Removal

There is no direct way of mapping a negated field into a single TCAM entry [22]. However, there is a scheme that we can use to replace negated fields with the `any` keyword. However as figure 4.2 illustrates, thoughtless application of negation removal would destroy the semantics of the classifier. Prior to negation removal, no packet could match both rule 1 and rule 2 because of the source port field. Yet after replacing rule 1's source port of `!50` with `any`, all packets matching rule 2 will match rule 1. This is a violation we cannot allow.

To remove negation and maintain classifier semantics, [22] presents a scheme for grouping rules of a classifier into *regions*. The regions logically partition the classifier. Each region $R$ is a set of rules with a domain $D(R)$ defined as a set of packets $P$ such that $\forall\ p \in P$, $\exists\ r \in R$ such that $p \subseteq r$. We want to reorder the rules so that all rules of each region are together and the regions are in decreasing order by number of negated fields.

To group rules into regions, the algorithm generates a special rule $s$ (which is called a *separator*) for each region such that $\forall\ p \in D(R), p \subseteq s$. By definition, each separator rule defines a region. The goal is that we place the separators at the end of their corresponding regions to stop packets that belong to the region from incorrectly matching later rules. Every separator has an empty matchlist, $m(s) = \varnothing$.

To generate separator rules, we must enumerate all negated fields in the classifier. Toward this end, we create $d$ sets of negated fields; one for each dimension of our classifier. For $\mathbb{F}_1, ..., \mathbb{F}_d$ we have a set of negated fields $\mathbb{N}_i$ for $1 \leq i \leq d$. For each rule $r \in \mathbb{AM}$, if $r[i]$ is negated we add the *non-negated* form of $r[i]$ to $\mathbb{N}_i$. For example, if $\mathbb{F}_3$ has the negated value `!45`, we add `45` to $\mathbb{N}_3$. After processing all rules, if a set $\mathbb{N}_i = \varnothing$, we append the `any` keyword to the set. After we have processed all fields of all rules, we generate the separator set $S$ by taking the $d$-ary Cartesian product of the $d$ negated field sets.

Our simple classifier presented above in figure 4.2 would generate just one separator because there is one distinct negated field in the classifier (`!50`). See figure 4.3 for a slightly more complex example where we generate three separator rules for two negated fields. We have a negated source port of `!80` and a negated destination port of `!50`, so we store `80` and `50` in the corresponding $\mathbb{N}_i$ negated field sets. Then we create the separators shown at the bottom of figure 4.3.

After we have generated the separator rules as described, we set $\mathbb{FM} = S$. That is, we

| Rule |
|---|
| TCP 70.10.32.41 !80 → 10.1.0.1 [10:20] |
| TCP 60.11.42.15 65 → 10.1.0.2 !50 |

↓

| Field | Value |
|---|---|
| Source port | 80 |
| Destination port | 50 |

↓

| Separators |
|---|
| any any 80 → any 50 |
| any any 80 → any any |
| any any any → any 50 |

Figure 4.3: Example of negation removal generating three separators for two negated fields.

initialize our first-match classifier to be the set of separators. Finally, we can apply the reduction of $\mathbb{AM}$ to $\mathbb{FM}$ as described in the preceding section. Our separators ensure that each rule in $\mathbb{AM}$ is inserted into the appropriate region. After we have transformed $\mathbb{AM}$ to $\mathbb{FM}$ we replace all negated fields in $\mathbb{FM}$ with the `any` keyword [22].

### 4.2.2 SSA

As noted in [22, 9, 23], the geometric all-match to first-match conversion used above generates many intersection rules which greatly increases the size of the resulting classifier. The Set Splitting Algorithm (SSA) of [23] reduces the number of intersection rules that we must store for the classifier. We now provide a brief overview of SSA.

SSA is an approach that seeks to minimize the number of additional intersection rules that need to be stored by splitting the classifier into multiple logically distinct sets. The key observation is that if we store the rules that compose an intersection in the different sets, we no longer need the intersection [23]. However, the cost of this solution is that we require two or four TCAM chips with 104-bit wide entries.

Thus, the problem is reduced to finding the best way to split the rules of $\mathbb{AM}$ into multiple sets. The ideal solution would split the rules such that any two rules that intersect are placed into different sets. In such as case, no new rules would be created. However, such a solution may not exist for all classifiers. In general, is equivalent to the NP-hard maximum set splitting or maximum hypergraph cut problem [4].

Given the problem is NP-hard, SSA uses Johnson's approximation algorithm [7] for the maximum satisfiability problem to split the classifier. We leave out the underlying details of Johnson's algorithm and only provide an overview of SSA. Nevertheless, let us restate the maximum satisfiability problem as presented by Yu et al. [23].

A *literal* $x_i$, is a variable (a *positive literal*) or the negation of a variable (a *negative literal*) to which we can assign either true or false. We are given a set of $n$ literal pairs: $L = \{\{x_1, \neg x_1\}, \{x_2, \neg x_2\}, ..., \{x_n, \neg x_n\}\}$. A *clause* is the disjunction of some set of literals. For example, we may have a clause $C = x_1 \vee \neg x_5 \vee \neg x_8$. By definition, a clause evaluates to true if any of its literals evaluate to true. For our problem, we are given $m$ clauses. The task at hand is to assign values to the literals in $L$ such that we satisfy as many clauses as possible.

We assume that we have applied the all-match to first-match reduction (with or without negation removal) and have the two classifiers $\mathbb{AM}$ and $\mathbb{FM}$. Let $\mathbb{I} \subseteq \mathbb{F}$ be the subset of $\mathbb{FM}$ containing all intersection rules generated. Furthermore, let $\mathbb{R} = \mathbb{FM} - \mathbb{I}$ be the subset of rules that are *not* intersection rules. Let $|\mathbb{R}| = n$ and $|\mathbb{I}| = m$. SSA proceeds as follows.

For each rule $r_i \in \mathbb{R}$, we create two literals: $x_i$ and $\neg x_i$. Next we create two clauses for each intersection $I \in \mathbb{I}$. Let $I$ be an intersection of $y$ rules: $I = r_1 \cap ... \cap r_y$. Then, the clauses we generate are $C_{I1} = \{r_1 \vee ... \vee r_y\}$ and $C_{I2} = \{\neg r_1 \vee ... \vee \neg r_y\}$ [23]. After we create the literals and clauses, we run Johnson's algorithm which assigns a value to each

| Rule # | Rule | Literals |
|---|---|---|
| 1 | TCP \$EXTERNAL_NET [10:30] → \$HOME_NET [10:30] | $x_1$, $\neg x_1$ |
| 2 | TCP \$EXTERNAL_NET [15:35] → \$HOME_NET [15:35] | $x_2$, $\neg x_2$ |
| 3 | TCP \$EXTERNAL_NET any → \$HOME_NET [5:20] | $x_3$, $\neg x_3$ |

↓

| Intersections | Rules | Clauses |
|---|---|---|
| TCP \$EXTERNAL_NET [15:30] → \$HOME_NET [15:20] | 1,2,3 | $(x_1 \vee x_2 \vee x_3)$ $(\neg x_1 \vee \neg x_2 \vee \neg x_3)$ |
| TCP \$EXTERNAL_NET [15:30] → \$HOME_NET [15:30] | 1,2 | $(x_1 \vee x_2)$ $(\neg x_1 \vee \neg x_2)$ |
| TCP \$EXTERNAL_NET [10:30] → \$HOME_NET [10:20] | 1,3 | $(x_1 \vee x_3)$ $(\neg x_1 \vee \neg x_3)$ |
| TCP \$EXTERNAL_NET [15:35] → \$HOME_NET [15:20] | 2,3 | $(x_2 \vee x_3)$ $(\neg x_2 \vee \neg x_3)$ |

↓

| Literal | Assignments |
|---|---|
| $x_1$ | False |
| $x_2$ | True |
| $x_3$ | False |

↓

| SSA Set 1 | SSA Set 2 |
|---|---|
| Rule 2 | Intersection of Rule 1 and Rule 3<br>Rule 1<br>Rule 3 |

Figure 4.4: Example of SSA splitting a set of rules into two sets.

literal. Finally we split $\mathbb{R}$ into two sets based on the literal assignments. If $x_i$ is true, we assign $r_i$ to set one; otherwise we assign $r_i$ to set two. Yu et al. [23] show that we only need to store an intersection, $I \subseteq \mathbb{I}$, if either $C_{I1}$ or $C_{I2}$ is unsatisfied. Figure 4.4 illustrates SSA on a set of three rules that have four intersections. SSA splits the seven rules into two sets with four total rules.

## 4.3 First-Match Classifier Optimization

After performing all-match to first-match conversion and processing our first-match classifier to be ready for additional minimization, we evaluated the following first-match optimization

algorithms: (1) TCAM Razor, (2) Topological Transformation, and (3) TCAM SPliT. Before we present an overview of each algorithm, we discuss two important topics: prefixes and firewall decision diagrams (FDD).

The first-match algorithms we apply work almost exclusively with prefixes. A $b$-bit *prefix* represents an integer interval and has $k$ leading 1's or 0's followed by $(b-k)$ $*$'s . That is, a prefix has the form $W\{*\}^{(b-k)}$ and represents the interval $[W\{0\}^{b-k}, W\{1\}^{b-k}]$ where $W \in \{0,1\}^k$. For example, the prefix, 101** represents the interval $[10100, 10111]$ or in decimal, $[20, 23]$ [2].

When storing rules in TCAM we must convert ranges into prefixes, which leads to the previously discussed range expansion problem. For example, the range $[2, 10]$ converts to the following *3*-bit prefixes, 01* and 1**.

All of the first-match algorithms we use convert the given classifier into a firewall decision diagram [10], which is a canonical representation of a classifier. A firewall decision diagram (FDD) represents a classifier and the set of decisions (denoted $DS$) of the classifier. An FDD is a directed, acyclic graph that has both node and edge labels. All FDDs must satisfy the following properties:

| Rule # | $F_1$ | $F_2$ | Decision |
|---|---|---|---|
| 1 | 10* | 10* | discard |
| 2 | 10* | 11* | accept |
| 3 | 10* | 0** | accept |
| 4 | 0** | *** | discard |
| 5 | 1** | *** | discard |

Table 4.1: Two-dimensional classifier where each field has domain $[0, 7]$.

[2]When working with an interval from $a$ to $b$ for first-match classifiers we use the common notation $[a, b]$, but when discussing Snort rules we use the Snort notation $[a : b]$.

1. There exists exactly one *root* node that has no incoming edges. We refer to all nodes with no outgoing edges as *terminals*, and all other nodes are *non-terminals*.

2. $F(v)$ is the label of node $v$:

   (a) $F(v) \in \{F_1, ..., F_d\}$ if $v$ is a non-terminal node.

   (b) $F(v) \in DS$ if $v$ is a terminal node.

3. $I(e)$ is a non-empty set of labels for edge $e : u \rightarrow v$ such that $I(e) \subseteq D(F(u))$. That is, $I(e)$ is a subset of the domain of the field corresponding to node $u$.

4. A path from the root to a terminal is called a *decision path*. All nodes on a decision path have a distinct label.

5. Two properties hold for $E(v)$, the set of outgoing edges of node $v$:

   (a) $I(e_1) \cap I(e_2) = \varnothing \ \forall \ (e_1, e_2) \in E(v)$. A prefix can match only one outgoing edge of $v$. This is called *consistency*.

   (b) $\bigcup_{e \in E(v)} I(e) = D(F(v))$. The union of the labels for all outgoing edges of node $v$ cover the domain of the field associated with $v$. This property is called *completeness*.

Let us illustrate the concept of an FDD with the classifier shown in table 4.1 and figure 4.5. The two-dimensional classifier is over fields $F_1$ and $F_2$ with $D(F_i) = [0, 7]$ for $1 \leq i \leq 2$. We can see how the FDD properties apply from the table and the figure. The following discussions present an overview of each of the first-match optimization algorithms we use.

After we construct an FDD for the classifier, we use an FDD reduction algorithm to make the FDD smaller, which reduces the number of rules the FDD generates. FDD reduction ensures three properties of the FDD: (1) no two nodes are isomorphic, (2) any pair of nodes

has at most one edge between them, and (3) no node has exactly one outgoing edge [5]. For example, we can reduce the FDD in figure 4.5 by removing the non-terminal right-child of the root, and instead have the outgoing edge go directly to the terminal node.

It is important to realize that the order in which we use the fields to build an FDD changes the FDD generated. This is an important consideration for two of the first-match classifier optimization algorithms that we use: TCAM Razor and TCAM SPliT. To maximize performance of these two algorithms we must try all *permutations* (orderings) of the fields when minimizing a classifier.

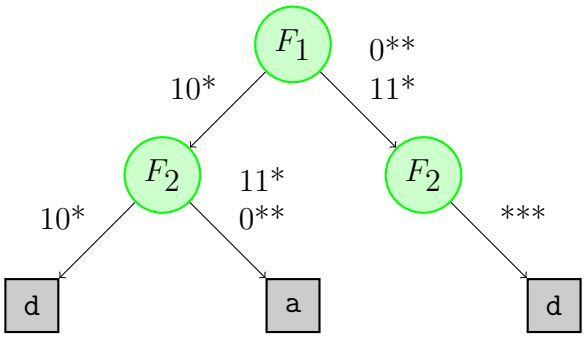


Figure 4.5: FDD corresponding to the classifier shown in table 4.1 with decisions "a" for *accept* and "d" for *discard*.

### 4.3.1 TCAM Razor

In [11], Liu et al. build on the dynamic programming solution presented in [19], with which Liu et al. solve the *weighted one-dimensional TCAM minimization problem*. The optimal one-dimensional solution is then extended to the multi-dimensional case to find a greedy solution for each dimension. Due to its greedy nature, Razor does not guarantee a globally optimal solution, but it is one of the best first-match minimization algorithms available. TCAM Razor is classified as an equivalent transformation algorithm because it produces a

| $F_1$ | $F_2$ | Decision |
|---|---|---|
| 101 | 11* | d |
| 101 | 10* | a |
| 101 | 0** | a |
| 110 | 11* | d |
| 110 | 10* | a |
| 110 | 0** | a |
| 111 | *** | d |
| 0** | *** | d |

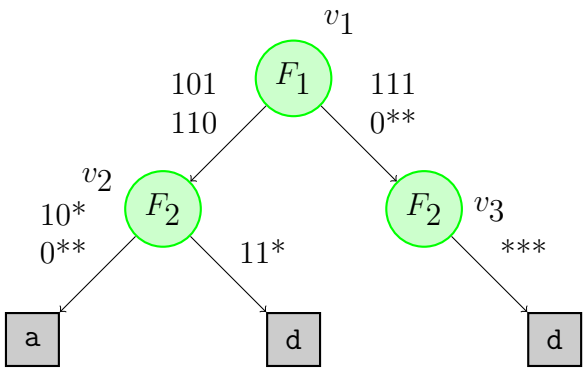


Figure 4.6: Two dimensional classifier with decisions and its associated FDD.

semantically equivalent, but smaller, classifier [11].

Given a classifier $\mathbb{C}$ with $d$ fields, $\mathbb{F}_1, ..., \mathbb{F}_d$, Razor converts the classifier into a firewall decision diagram (FDD). Before minimizing the classifier, Razor next reduces the FDD. Following FDD creation and reduction, Razor builds a classifier $\mathbb{C}'$ that is semantically equivalent to $\mathbb{C}$ in a bottom-up fashion by repeatedly applying the optimal one-dimensional solution to obtain a solution for each field. These solutions are combined to give an overall solution. Finally, Razor applies a redundancy removal algorithm to $\mathbb{C}'$ to remove redundant rules.

Consider the two-dimensional classifier and the associated FDD shown in 4.6. For Razor, we use the one-dimensional minimization algorithm on both subgraphs rooted at nodes $v_2$ and $v_3$ to get the TCAM tables shown in 4.7.

Next we consider $v_2$ and $v_3$ to be decisions and process $v_1$ as if it were a one-dimensional node and we obtain the resulting TCAM table in 4.8. Finally we put the three tables together to obtain to the final TCAM table in 4.9. The resulting TCAM table has four rules while the original classifier had eight rules.

### 4.3.2 Topological Transformation

Topological Transformation [14] is a range encoding scheme for minimizing first-match classifiers. Unlike other range encoding schemes, Topological Transformation uses rule decisions when minimizing the classifier. The algorithm's key observation is that many values for each field are equivalent; two values are said to be equivalent if they can be interchanged without changing the decision for a packet. A classifier partitions each field into equivalence classes such that all values with the same decision belong to the same class. The goal is to reduce the number and size of TCAM entries by eliminating equivalent values.

Like TCAM Razor, Topological Transformation first converts a classifier to an FDD. Then we apply two reduction techniques, *domain compression* and *prefix alignment*. Domain compression attempts to make the domain of each field, $D(F_i)$, as small as possible, while prefix alignment modifies the domain of each field so that we can convert ranges to prefixes with less expansion [14]. Domain compression uses three steps: (1) first we compute the equivalence classes for each field, (2) we build *transformers* for each field, (3) we apply the field transformers to create the re-encoded classifier [14].

For a $d$-dimensional classifier, we compute the equivalence classes for each field $\mathbb{F}_i$ by first creating an FDD, $f_i$, rooted at $\mathbb{F}_i$ for $1 \leq i \leq d$. Let $u$ be the root of the FDD $f_i$ and let $u$ have outgoing edges $e_1, ..., e_m$. Meiner's et al. [14] observe that the labels of these outgoing edges form the equivalence classes for $\mathbb{F}_i$. Each label is associated with several ranges, but

| $v_2$ | | $v_3$ | |
|---|---|---|---|
| $F_2$ | Decision | $F_2$ | Decision |
| 11* | d | *** | d |
| *** | a | | |

Figure 4.7: One-dimensional solutions for nodes $v_2$ and $v_3$.

| $v_1$ | |
|---|---|
| $F_1$ | Decision |
| 111 | $v_3$ |
| 1** | $v_2$ |
| *** | $v_3$ |

Figure 4.8: One-dimensional solution for node $v_1$ with virtual $v_2$ and $v_3$ nodes.

| $F_1$ | $F_2$ | Decision |
|---|---|---|
| 111 | *** | d |
| 1** | 11* | d |
| 1** | *** | a |
| *** | *** | d |

Figure 4.9: Result of TCAM Razor on the classifier from 4.6.

all ranges belong to the same equivalence class. Thus we choose the range that intersects the fewest rules in $\mathbb{C}$ to represent the equivalence class for each edge $e_j$. Once a range, denoted by $L_m$, has been chosen for each edge, we put the edges in ascending order by the range values $L_m$. After we sort the edges, we have the transformer $\mathbb{T}_i$ for field $\mathbb{F}_i$, which assigns edge $e_j$ the decision $j$ for $1 \leq j \leq m$. Next we can use these transformers to create the transformed classifier.

To create the transformed classifier $\mathbb{C}'$ for $\mathbb{C}$ we must re-encode each rule using the transformers we computed in the preceding step. Let $S_i = [x, y]$ be the original range for field $\mathbb{F}_i$. $S_i' = [a, b]$ where a is $\arg\min_a a \in [0, m-1]$ such that $L_a \cap S_i \neq \varnothing$ and b is $\arg\max_b b \in [0, m-1]$ such that $L_b \cap S_i \neq \varnothing$. If a range in $\mathbb{C}$ does not have an intersection with any landmark, we do not create a transformed rule in $\mathbb{C}'$. That is, for any rule $r \in \mathbb{C}$, if $\exists$ a field $\mathbb{F}_i$ where $r[i]$ has an empty intersection with all landmark ranges in $\mathbb{F}_i$, then we do not create the transformed rule $r'$ in $\mathbb{C}'$.

The second optimization that Topological Transformation provides is with prefix alignment. Prefix alignment again re-encodes the domain of each field, but the goal now is to change the field domains to be such that ranges can be encoded with fewer prefixes.

Domain compression is illustrated in figures 4.10 to 4.13. Figure 4.10 presents a two-dimensional classifier we want to perform domain compression on. Figure 4.11 shows the

| $F_1$ | $F_2$ | Decision |
|---|---|---|
| $[1, 9]$ | $[2, 26]$ | d |
| $[29, 31]$ | $[2, 26]$ | d |
| $[0, 31]$ | $[0, 19]$ | a |
| $[12, 17]$ | $[19, 28]$ | d |
| $[0, 30]$ | $[4, 27]$ | a |
| $[0, 31]$ | $[0, 31]$ | a |

Figure 4.10: Two dimensional classifier expressed with ranges.

| $F_1$ |
|---|
| $[0, 0]$ |
| $[1, 9]$ |
| $[10, 11]$ |
| $[12, 17]$ |

Figure 4.11: Landmarks identified by Topological Transformation for field $F_1$.

| $F_1$ | Decision |
|---|---|
| $[0, 0]$ | 0 |
| $[1, 9]$ | 1 |
| $[10, 11]$ | 2 |
| $[12, 17]$ | 3 |
| $[18, 28]$ | 2 |
| $[29, 31]$ | 1 |

Figure 4.12: Transformer for field $F_1$.

| $F_1$ | $F_2$ | Decision |
|---|---|---|
| $[1, 1]$ | $[0, 1]$ | d |
| ~~$[1, 1]$~~ | ~~$[0, 1]$~~ | ~~d~~ |
| $[0, 3]$ | $[0, 2]$ | a |
| $[3, 3]$ | $[0, 1]$ | d |
| $[0, 3]$ | $[0, 1]$ | a |
| $[0, 3]$ | $[0, 2]$ | a |

Figure 4.13: Transformed classifier.

landmark ranges identified after Topological Transform creates an FDD rooted at field $F_1$. Figure 4.12 lists the transformer for the ranges in field $F_1$. Figure 4.13 shows the transformed classifier, which clearly contains redundant rules. The crossed-out rule in figure 4.13 indicates that the rule would not be generated because $[29, 31]$ does not intersect any landmark ranges for $F_1$.

### 4.3.3 TCAM SPliT

TCAM SPliT [15] is a first-match optimization algorithm that splits a classifier into several smaller classifiers. SPliT is intended for use in an environment where multiple TCAM chips can be pipelined to perform packet classification. The observation exploited by SPliT is that higher dimensional classifiers suffer from the *multiplicative effect* where rules interact badly and cause additional TCAM entires. By splitting a classifier into multiple lower dimensional

classifiers, SPliT avoids the cost of the multiplicative effect [15].

SPliT first converts a classifier $\mathbb{C}$ into an FDD. Then we reduce the FDD by merging all isomorphic subgraphs [14]. Following this setup phase, SPliT applies a divide and conquer approach to create several smaller classifiers from $\mathbb{C}$. The two steps SPliT uses to create smaller classifiers are: (1) node table generation, and (2) field table generation.

During *node table generation*, SPliT creates a one-dimensional TCAM table for each non-terminal in the FDD. Each of these tables has one rule per prefix, per outgoing edge. The decision SPliT assigns each of these rules is the ID of the node to which the outgoing edge goes to.

*Field table generation* combines the node tables for each field we created in the previous step into one TCAM table. This involves three steps. First we assign a unique node ID to each of the node tables. Next we prepend the ID for each node table to all the rules contained in the table. Lastly, SPliT combines all tables for each field into one TCAM table.

We can optimize field table generation by using *shadow encoding* when combining the tables for field $\mathbb{F}_i$ [13]. While FDD reduction combines isomorphic nodes together, there are often nodes that are not isomorphic, but similar. Shadow encoding leverages the similarity among nodes during field table generation using two key observations: we have free reign over what table IDs to assign and we can use ternary strings. Shadow encoding allows us to remove similar entries from one node table and use a ternary string for the table ID of another table to match both tables, thus reducing the overall size of the field table [15].

In the end, SPliT produces multiple classifiers of smaller dimension than $\mathbb{C}$. Typically, SPliT will produce either $d$ one-dimensional classifiers or two classifiers, one with $k$ dimensions and the other with $(d - k)$ dimensions. TCAM SPliT is a state-of-the-art first match optimization algorithm when multiple TCAM chips can be used [15].

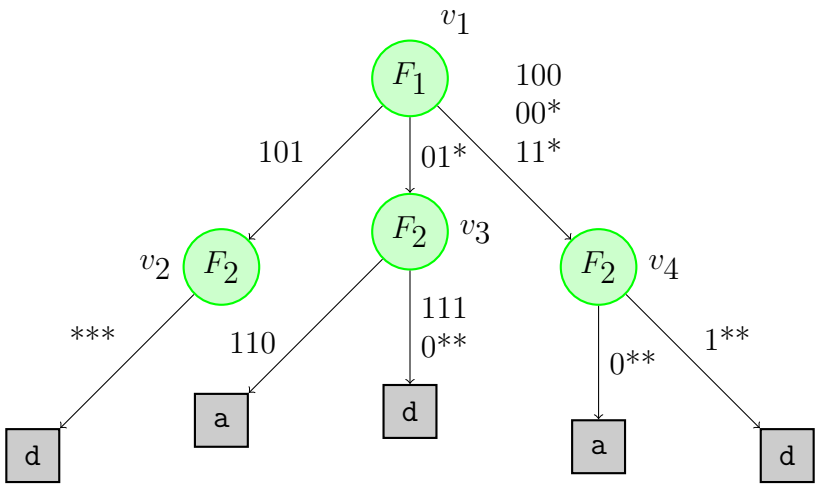


Figure 4.14: FDD for node and field table generation.

We assign nodes $v_2$, $v_3$, and $v_4$ node IDs 00, 01, and 11, respectively. Then we can create the field table for $F_2$. We prepend the node ID of a node table to its rules when we add them to the field table.

To illustrate shadow encoding, consider the field table composed of two node tables ($t_1$ and $t_2$) shown in figure 4.16. We can see that $t_1$ and $t_2$ are identical except that $t_1$ has one additional rule. We let $t_1$ defer to $t_2$ and we get the shadow encoded field table in 4.16

## 4.4 Verification

Our work would be pointless without verification that the first-match classifier resulting from the all-match to first-match conversion was actually equivalent to the original all-match classifier. To our knowledge there is no algorithm short of brute force to confirm that an all-match classifier $\mathbb{AM}$ and a first-match classifier $\mathbb{FM}$ are equivalent. We verified classifier equivalence by generating a comprehensive set of packets and verifying both classifiers produced the same result for each packet.

We generate two sets of packets `InputPackets` and `OutputPackets` based on $\mathbb{AM}$ and

| $v_1$ node table | |
|---|---|
| 101 | 00 |
| 01* | 01 |
| *** | 11 |

| $v_2$ node table | | $v_3$ node table | | $v_4$ node table | |
|---|---|---|---|---|---|
| *** | d | 110 | a | 0** | a |
| | | *** | d | *** | d |

| $F_2$ Field table | |
|---|---|
| 00 *** | d |
| 01 110 | a |
| 01 *** | d |
| 11 0** | a |
| 11 *** | d |

Figure 4.15: Node tables for $F_1$ and $F_2$, and field table for $F_2$.

| | | | | | | |
|---|---|---|---|---|---|---|
| $t_1$ | 00 110 | a | → | $t_1$ | 00 100 | a |
| | 00 100 | a | | $t_2$ | 0* 110 | a |
| | 00 *** | d | | | 0* *** | d |
| $t_2$ | 01 110 | a | | | | |
| | 01 *** | d | | | | |

Figure 4.16: Field table and shadow encoded field table for $F_2$.

$\mathbb{FM}$, respectively. However, since the packet generation procedure is the same for both classifiers, let us focus on the packet generation for $\mathbb{AM}$.

Recall that each field in $\mathbb{AM}$ can consist of several discontinuous intervals and the $i^{th}$ field of rule $r$ is $r[i]$. For each interval $int = (s, e)$ of $r[i]$ for each rule $r \in \mathbb{AM}$, generate four packets $p_j$ for $1 \leq j \leq 4$ as follows: (1) $p_j[k] \in \mathbb{F}_k \ \forall \ k \neq i$. (2) $p_j[i]$ uses the values $(s-1,\ s,\ e,\ e+1)$ for $1 \leq j \leq 4$, respectively. For example, Let $r$ be a rule with a source port field of [25:30,35:40], then we generate eight packets (four for each interval) with source port values: 24, 25, 30, 31, 34, 35, 40, and 41.

After using the above method to generate the `InputPackets` and `OutputPackets` sets for

$\mathbb{AM}$ and $\mathbb{FM}$, we cross-classified the packets against both classifiers to ensure that neither classifier made misclassifications.

# Chapter 5: Experimental Results

In this chapter, we present our experimental results. We evaluated three versions of the Snort rules using our two approaches: (1) applying all-match classifier specific optimization algorithms and (2) transforming the given all-match classifier to a first-match classifier and then using state-of-the-art first-match classifier optimization algorithms.

When optimizing for a one TCAM environment, our results indicate it is best to perform first-match classifier optimization on a transformed classifier when we perform no all-match specific optimizations. Using TCAM Razor, we had TCAM space savings of nearly 71%. However, when optimizing for a multiple TCAM environment, our results indicate using SSA with Negation Removal is most effective.

First we discuss some pertinent implementation details. Then we present the classifiers we used and what effect the all-match to first-match conversion has on them. We end by giving performance and efficiency results for both approaches.

## 5.1 Methods

In our experiments, we used the three Snort rule sets shown in column one of table 5.1. The Snort rules often used values like `$HOME_NET` and `$EXTERNAL_NET` for the source IP and destination IP fields. In our experiments, we set `$HOME_NET` to be a full class A subnet. By default, we defined `$EXTERNAL_NET` to be the negation of `$HOME_NET`. We did test different initializations for `$HOME_NET`, but noticed no significant effect on the results.

We implemented the all-match classifier to first-match classifier transformation algorithm, the negation removal algorithm, and the Set Splitting Algorithm (SSA). In addition, we used previous implementations of first-match optimization algorithms TCAM Razor, Topological

Transformation, and TCAM SPliT.

For TCAM Razor and TCAM SPliT, the order in which we process the fields of the classifier has an impact on the results. We evaluated all permutations of field ordering and have reported the results for the best ordering for each algorithm. To compare classifier efficiency, we averaged the running times of the five best permutations of field ordering for each classifier and algorithm.

While evaluating first-match optimizations for our classifiers, we considered different values for the width of each TCAM entry. We evaluated using a perfect TCAM width of 104 bits, where each TCAM entry is exactly as wide as each rule, as well as multiples of 36 and 40bit TCAM entry widths. Specifically, we focused on TCAM widths of 104, 144, and 160 bits. The 104 bit perfect TCAM width is the sum of the width of each field; we assume the protocol is an 8 bit field, the source and destination IP fields are 32 bits each, and the source and destination port fields are 16 bits each. While having a perfect width TCAM entry in practice is unrealistic, it provides a lower bound on the size of our classifiers.

Negation removal, SSA, and TCAM Razor all use 144 or 160 bit wide TCAM entries because each rule is 104 bits wide. Thus, we are wasting 40 or 56 bits per entry, respectively. For TCAM SPliT and Topological Transformation transformers, we also considered TCAM widths of 36, 40, 72, and 80 bits since each TCAM entry only encodes one field plus a table ID.

To evaluate the effectiveness of each algorithm, we compute the TCAM space saved compared to our baseline result using only negation removal. Specifically, we calculate TCAM space saved as,

$$\text{TCAM Space Saved} = 1 - \frac{\text{TCAM Size of Optimized Classifier}}{\text{TCAM Size of Classifier with only Negation Removal}}.$$

| | | Negation Removal | | w/o Negation Removal | |
|---|---|---|---|---|---|
| Rules | No. Rules | Output Rules | Converted Rules | Output Rules | Converted Rules |
| 2.9.0.3 | 471 | 19240 | 90279 | 18685 | 177662 |
| 2.9.0.5 | 552 | 28239 | 176562 | 27564 | 349809 |
| 2.9.1.0 | 553 | 28243 | 176566 | 27565 | 349810 |

Table 5.1: Number of input and output rules through the all-match to first-match conversion.

| | Negation Removal | w/o Negation Removal |
|---|---|---|
| Rules | TCAM Entries | TCAM Entries |
| 2.9.0.3 | 131043 | 1022184 |
| 2.9.0.5 | 246276 | 1939700 |
| 2.9.1.0 | 246282 | 1939701 |

Table 5.2: Number of input and output rules through the all-match to first-match conversion.

More precisely, the TCAM space saved results we present are *in addition to* the inherent savings given in [22] when performing negation removal during all-match to first-match conversion.

## 5.2 Baseline Results

First we discuss how the classifiers expand as we progress through the all-match to first-match conversion process. As we can see in table 5.1, the number of rules increases dramatically when we perform the all-match to first-match conversion. The "Output Rules" columns of table 5.1 indicate how many rules the all-match to first-match algorithm produced using or not using negation removal. The "Converted Rules" columns indicate the number of rules in the each classifier after splitting rules with discontinuous ranges. Table 5.2 shows the number of TCAM entries required to store the converted rules without further optimization. We can compute the size of these classifiers by multiplying by the number of bits used per entry.

The all-match to first-match algorithm in [22] increases the size of the classifier. This is quite obvious when comparing "No. Rules" column to the values shown in any of the output columns. In addition, the "Output Rules" are greater when we perform negation removal. This is to be expected because the separator rules we add to maintain classifier semantics will generate new intersection rules with other rules.

At first glance it may be puzzling why the "Converted Rules" are greater when we do not perform negation removal, but there is a simple explanation for this phenomenon. When we skip negation removal, the rules with a field of `$EXTERNAL_NET` have discontinuous ranges that we need to split into multiple rules. Since most of the rules have `$EXTERNAL_NET` for one of the IP fields, we effectively double the size of the classifier.

### 5.2.1 Single TCAM Approach

We first evaluated the TCAM Razor first-match classifier optimization algorithm [11]. The percent of TCAM space saved using TCAM Razor on the first-match classifier for Snort 2.9.0.3 was 56.49%, as can be seen in Figure 5.1. On the newer and larger Snort rule sets 2.9.0.5 and 2.9.1.0, Razor achieved space savings of 70.85%, as can be seen in Figure 5.2. Because TCAM Razor uses an FDD while minimizing classifiers, we attain the same TCAM space savings whether or not we use negation removal during the all-match to first-match conversion.

### 5.2.2 Mutliple TCAM Approach

When using multiple TCAM chips is an option, our results indicate it is best to use the all-match specific optimizations negation removal and SSA instead of the first-match classi-

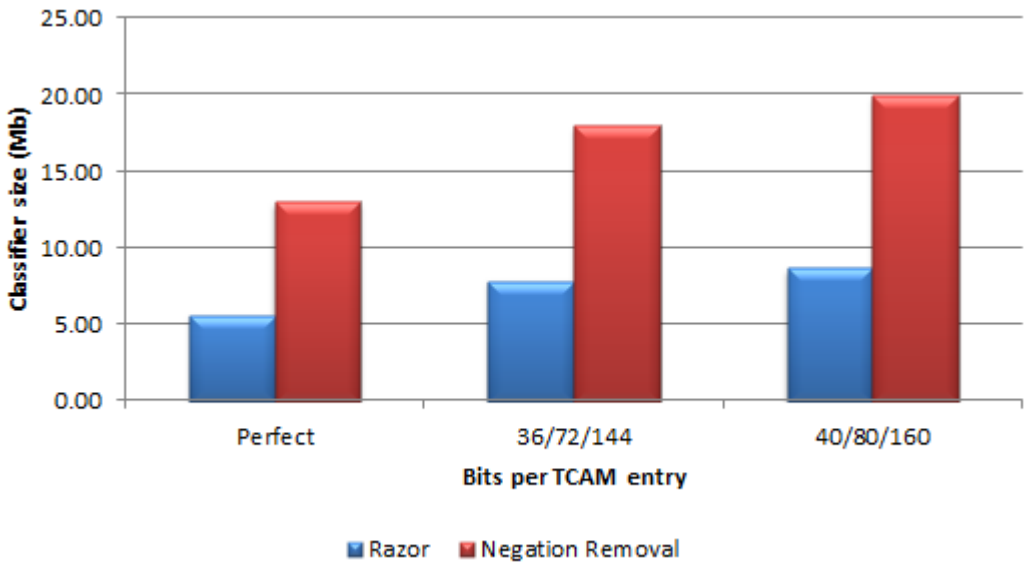


Figure 5.1: Classifier sizes obtained using TCAM Razor vs negation removal alone for the Snort 2.9.0.3 rule set.

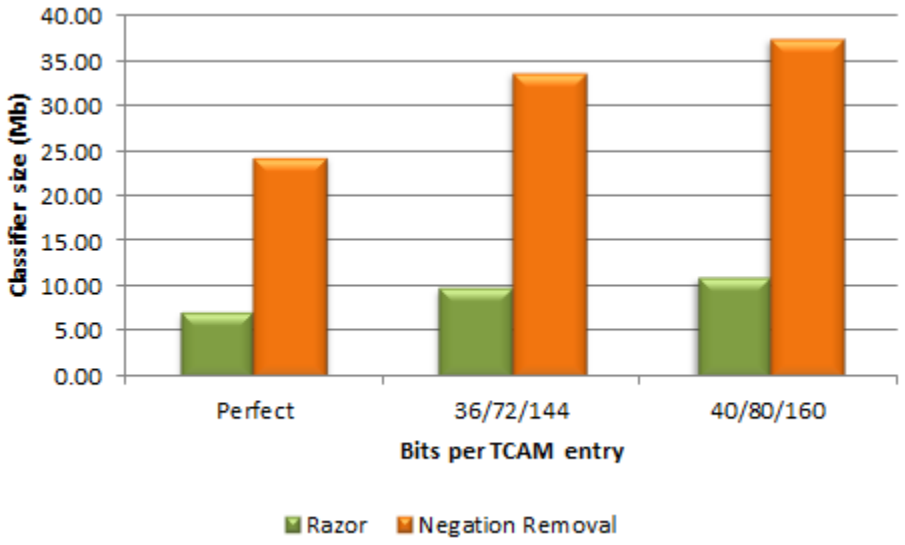


Figure 5.2: Classifier sizes obtained using TCAM Razor vs negation removal alone for the Snort 2.9.0.5 and 2.9.1.0 rule sets.

| | | Snort Rules: | 2.9.0.3 | 2.9.0.5 | 2.9.1.0 |
|---|---|---|---|---|---|
| Algorithm | Bits | TCAM | TCAM Entries | | |
| Negation Removal | 144 | Entries | 131043 | 246276 | 246282 |
| | | Size | 18.00 Mb | 33.82 Mb | 33.82 Mb |
| | | Space Savings | — | — | — |
| TCAM Razor | 144 | Entries | 57018 | 71798 | 71799 |
| | | Size | 7.83 Mb | 9.86 Mb | 9.86 Mb |
| | | Space Savings | 56.49% | 70.85% | 70.85% |
| Topological Transformation | 40 | Entries | 28491 | 36978 | 36979 |
| | | Size | 1113 Kb | 1444 Kb | 1444 Kb |
| | | Space Savings | 93.96% | 95.83% | 95.83% |
| TCAM SPliT | 36 | Entries | 26465 | 33952 | 33952 |
| | | Size | 931 Kb | 1194 Kb | 1194 Kb |
| | | Space Savings | 94.95% | 96.55% | 96.55% |
| SSA 2 | 144 | Entries | 6518 | 1948 | 1432 |
| | | Size | 917 Kb | 274 Kb | 201 Kb |
| | | Space Savings | 95.03% | 99.21% | 99.42% |
| SSA 4 | 144 | Entries | 1149 | 1323 | 1188 |
| | | Size | 162 Kb | 186 Kb | 167 Kb |
| | | Space Savings | 99.12% | 99.46% | 99.52% |

Table 5.3: Summary of results.

fier optimizations Topological Transformation and SPliT. SSA achieves better performance because it solves a slightly different problem. While Topological Transformation and TCAM SPliT return all matching rules, SSA returns only a subset per query. The results returned by all SSA lookups must be unioned together to get a complete solution.

Table 5.3 summarizes the performance of all algorithms we evaluated. We have omitted space savings for negation removal because it was our baseline, but in fact the space savings for negation removal can be calculated by comparing against the size of the classifier if we do not apply negation removal.

We can see that SSA 2 and SSA 4 outperformed both Topological Transformation and TCAM SPliT. Note that of the multi-TCAM optimization algorithms, Topological Transformation had the least space savings with 93.96% and SSA 4 had the most with 99.52%.

## 5.3 Efficiency

The computational effort required to compress large classifiers like the Snort rules is a non-trivial task. In this section, we evaluate the efficiency of our approaches. Surprisingly, we report an increase in efficiency when we skip negation removal during the all-match to first-match conversion.

We only collected efficiency data for TCAM Razor and Topological Transformation; however, Topological Transformation had an average run time on the order of hours while SSA 2 and SSA 4 were on the order of 10's of minutes. Therefore, we only present efficiency results for TCAM Razor.

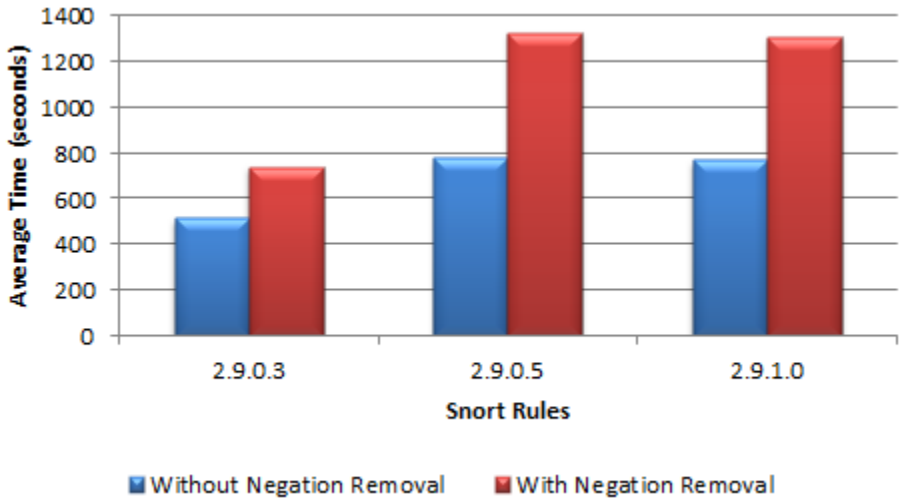


Figure 5.3: Time efficiency of TCAM Razor with and without negation removal.

As figure 5.3 shows, we found that on average Razor performed 38% faster for input classifiers generated without using negation removal. This is a meaningful result given that the classifiers without negation removal are twice as large as the classifiers with negation removal (see table 5.1). Since the two classifiers are semantically equivalent, they are represented by

the same FDD. Therefore, the only explanation for Razor being more efficient on one or the other is that Razor can build the FDD faster using the classifier formed without negation removal.

# Chapter 6: Conclusion

In this thesis, we evaluate the performance and efficiency of two approaches to minimizing all-match classifiers by converting them to first-match classifiers. In the first approach we, consider optimization algorithms specific to all-match classifiers: negation removal and SSA. Our second approach applies state-of-the-art first-match classifier optimization algorithms to a first-match classifier generated from an all-match classifier. We evaluated both approaches in the context of how many TCAM chips are used. We found that when only one TCAM chip is available, it is best to avoid all-match specific optimizations and apply first-match classifier optimizations instead. Using this approach we were able to achieve a 70.85% reduction in TCAM space with TCAM Razor over negation removal alone. On the other hand, we found in a multiple TCAM chip environment, the best method to use is SSA with negation removal which achieves TCAM space savings of 99.52% over negation removal alone.

# BIBLIOGRAPHY

# BIBLIOGRAPHY


[1] Snort. `http://www.snort.org/`.

[2] A. Bremler-Barr and D. Hendler. Space-efficient tcam-based classification using gray coding. *Computers, IEEE Transactions on*, 61(1):18 –30, jan. 2012.

[3] Hao Che, Zhijun Wang, Kai Zheng, and Bin Liu. Dres: Dynamic range encoding scheme for tcam coprocessors.

[4] Pierluigi Crescenzi and Viggo Kann. A compendium of np optimization problems. `http://www.nada.kth.se/~viggo/wwwcompendium/node145.html`.

[5] Mohamed G. Gouda and Alex X. Liu. Structured firewall design. *Comput. Netw.*, 51(4):1106–1120, March 2007.

[6] P. Gupta and N. McKeown. Algorithms for packet classification. *Network, IEEE*, 15(2):24 –32, mar/apr 2001.

[7] David S. Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, STOC '73, pages 38–49, New York, NY, USA, 1973. ACM.

[8] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for advanced packet classification with ternary cams. In *In ACM SIGCOMM*, pages 193–204, 2005.

[9] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for advanced packet classification with ternary cams. In *In ACM SIGCOMM*, pages 193–204, 2005.

[10] Alex X. Liu and Mohamed G. Gouda. Diverse firewall design. *Parallel and Distributed Systems, IEEE Transactions on*, 19(9):1237 –1251, sept. 2008.

[11] A.X. Liu, C.R. Meiners, and E. Torng. Tcam razor: A systematic approach towards minimizing packet classifiers in tcams. *Networking, IEEE/ACM Transactions on*, 18(2):490 –500, april 2010.

[12] Huan Liu. Efficient mapping of range classifier into ternary-cam. In *High Performance Interconnects, 2002. Proceedings. 10th Symposium on*, pages 95 – 100, 2002.

[13] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. Fast regular expression matching using small tcams for network intrusion detection and prevention systems. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[14] C.R. Meiners, A.X. Liu, and E. Torng. Topological transformation approaches to tcam-based packet classification. *Networking, IEEE/ACM Transactions on*, 19(1):237 –250, feb. 2011.

[15] C.R. Meiners, A.X. Liu, E. Torng, and J. Patel. Split: Optimizing space, power, and throughput for tcam-based classification. In *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, pages 200 –210, oct. 2011.

[16] D. Pao, P. Zhou, B. Liu, and X. Zhang. Enhanced prefix inclusion coding filter-encoding algorithm for packet classification with ternary content addressable memory. *IET Computers & Digital Techniques*, 1(5):572–580, 2007.

[17] Derek Pao, Yiu Keung Li, and Peng Zhou. Efficient packet classification using tcams. *Computer Networks*, 50(18):3523 – 3535, 2006.

[18] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using extended tcams. In *Network Protocols, 2003. Proceedings. 11th IEEE International Conference on*, pages 120 – 131, nov. 2003.

[19] Subhash Suri, Tuomas Sandholm, and Priyank Warkhede. Compressing two-dimensional routing tables, 2003.

[20] David E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37(3):238–275, September 2005.

[21] Pi-Chung Wang and Chia-Ming Chang. Scalable packet classification for network intrusion detection. In *Proceedings of the Fifth IASTED International Conference on Circuits, Signals and Systems*, CSS '07, pages 64–69, Anaheim, CA, USA, 2007. ACTA Press.

[22] F. Yu, R.H. Katz, and T.V. Lakshman. Efficient multimatch packet classification and lookup with tcam. *Micro, IEEE*, 25(1):50 – 59, jan.-feb. 2005.

[23] F. Yu, T.V. Lakshman, M.A. Motoyama, and R.H. Katz. Efficient multimatch packet classification for network security applications. *Selected Areas in Communications, IEEE Journal on*, 24(10):1805 –1816, oct. 2006.