DIVIDE AND CONQUER: PACKET CLASSIFICATION BY SMART DIVISION

By

James Edward Daly

A DISSERTATION

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

Computer Science — Doctor of Philosophy

2017

ABSTRACT

DIVIDE AND CONQUER: PACKET CLASSIFICATION BY SMART DIVISION

By

James Edward Daly

Packet classifiers form the backbone of many networking and security devices. Most packet classifiers are defined by a list of rules that each match a subset of all packets. As new threats and vulnerabilities emerge, these rule lists are getting larger and thus taking longer to search. However, networks have very strict time constraints and delays classifying a packet at one stage can cascade and cause delays and congestion throughout the network. Additionally, with the advent of software-defined networks, these rule lists are also constantly changing. Packet classifiers thus must support both fast classification time and fast updates.

We present four methods to improve packet classification. Each method uses smart division to divide the problem into simpler subproblems that are more easily solved. Diplomat uses dimensional reduction to create a smaller rule list by dividing the problem into several lower dimensional problems and then combines them with as few rules as possible. ByteCuts uses rule list separation to partition the rule list into a small number of trees; this minimizes the rule replication that other tree-based methods such as HyperCuts and HyperSplit require. PartitionSort uses rule list separation to partition the rule list into parts that are sortable; each part can then be binary searched. Finally, TupleMerge uses rule list separation to partition the rule list into parts that are hashable; it requires fewer hash tables than Tuple Space Search which reduces search times.

This thesis is dedicated to my wife Couri and the rest of my family, who have supported me throughout the way.

ACKNOWLEDGMENTS

I am eternally grateful to Dr. Eric Torng who served as my advisor, guide, and editor throughout this process. None of this would have been possible without his help. I would also like to thank Dr. Alex Liu who provided additional guidance. I would also like to recognize the rest of my committee, Dr. Abdol-Hossein Esfahanian and Dr. Jian Ren for their insights and contributions.

I would also like to recognize the contributions of Dr. Chad Meiners, whose code I inherited, and Sorrachai Yingchareonthawornchai, who helped with some of the coding and testing. This would have been even longer and more difficult without their contributions.

Finally, I would like to thank my family for encouraging me in this endeavor, especially my wife Couri and my parents, who were beside me every step of the way.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

KEY TO SYMBOLS

c An action or color

x, y Packets

r A rule

s An interval

p A prefix interval

w Prefix length

d Number of dimensions

n Number of rules

n_i Number of rules in set ℓ_i

m_i Length of field i

$\mathbb{C}$ Packet domain

$\mathcal{K}$ Set of actions

$P : \mathbb{C} \to \mathcal{K}$ A pattern

$P(x)$ The color of packet x

L A rule list

ℓ A partial rule list

Chapter 1

Introduction

Routers, firewalls, and other devices perform a variety of network services, such as packet filtering, network address translation (NAT), load balancing, and traffic monitoring. These devices receive a stream of packets from the network and must decide how each packet should be handled. These devices use a packet classifier to determine this behavior. While different types of services have different requirements, in general, the classifier looks at a d-tuple from fields in the packet header to make some sort of decision.

Network devices have real-time constraints. They continuously receive packets which they need to respond to promptly; a delay in processing one packet can result in delays in subsequent packets. Additionally, queue sizes are limited. If processing takes too long, the queue will fill up, resulting in dropped packets. This contributes significantly to network congestion and should be avoided. Fast packet classification thus reduces the number of queued packets and reduces congestion.

1.1 Offline Packet Classification

In traditional offline packet classification, each classifier is defined by a list of n rules that each map a set of packet headers to some decision. Usually each rule will use some subset of the following 5 fields: source address, destination address, source port, destination port, and protocol. Since there is no guarantee that the sets are disjoint, if multiple rules match

a packet, the first matching rule carries precedence.

Once a rule list is created, it is relatively static with infrequent changes. Thus, it is sufficient to create a single static structure that can search the rule list as fast as possible. The fastest methods can classify a packet in $O(\log n)$ time, but they require $O(n^d)$ memory [28]. Unfortunately, memory is not an infinite resource and this is too large for all but the smallest classifiers. Other methods require only $O(n)$ memory but have $O(\log^d n)$ search times. As such, packet classification involves trade-offs between time and memory.

From the above, the *Offline Packet Classification* problem is as follows: given a rule list L and a set of memory and other resource constraints, construct a data structure where the expected time to find the first matching rule for each packet in a stream is minimized.

1.2 Online Packet Classification

OpenFlow and other software defined networks add multiple difficulties to traditional packet classification [17]. First, the rule list is often updated by a controller, sometimes multiple times a second. As such, it is no longer allowable for the classifier to be a static entity. Instead, it must be able to be quickly updated. Furthermore, additional packet fields are defined for network behavior and the controller can define other arbitrary fields. Thus, the classifier used must be able to scale to a larger numbers of fields.

The *Online Packet Classification Problem* thus features revised requirements from the offline problem: given a (many-dimensional) rule list L, a time allowance τ, and a set of memory and other resource constraints, construct a data structure which can be updated in under τ time and the expected time to find the first matching rule for each packet in a stream is minimized.

1.3 Strategy Overview

Packet classification is a difficult problem. In this thesis, we present four methods which use a strategy which we call smart division. In each method, the problem is divided into multiple simpler problems. We then create a solution for each subproblem. These solutions are then recombined to create a solution for the original problem.

We use two types of smart division. In dimensional reduction, a higher dimensional problem is reduced to a series of lower-dimensional problems. This process is repeated until we get 1-dimensional problems for which good solutions exist. These subsolutions are then recombined to form a solution to the original problem. For rule list separation, the original rule list is divided into a series of sublists, each of which has certain desirable properties. We can then create a classifier for each of these sublists which are then combined to form a classifier for the original rule list.

1.4 Organization

The rest of this thesis is organized as follows. In Chapter 2, we review a variety of prior work in this field, grouped according to the taxonomy used by Taylor in [28]. In Chapter 3, we present definitions and terminology used in common between all of the sections. In Chapter 4, we describe Diplomat, a dimensional reduction method for compressing rule lists into smaller lists. In Chapter 5, we describe ByteCuts, a rule list separation method that creates multiple decision trees and focuses on fast search times. In Chapter 6, we describe PartitionSort, a rule list separation method that supports fast updates by creating multiple search trees by imposing a partial ordering over the rules. In Chapter 7, we describe TupleMerge, a rule list separation method that supports fast updates by separating the rules into multiple hash

tables. Finally, in Chapter 8, we offer some concluding remarks and identify some open problems.

Chapter 2

Review of Current Topics

This section covers currently existing methods. It is divided into four sections according to the Taylor taxonomy [28].

2.1 Exhaustive Search

This category covers both sequential search and TCAMS. In a sequential search, each rule is checked individually until a matching rule is found. This becomes very expensive time-wise as the number of rules increases as it requires $O(n)$ time.

TCAMs are special hardware chips that allow the entire list to be searched in parallel. However, their memory capacity has not scaled at the rate predicted by Moore's Law [14]. Additionally, they tend to be expensive and consume a large amount of power. As such, the number of rules that can be used in TCAMs is very limited.

Since both sequential search and TCAMs benefit significantly from smaller rule lists, research in this area tends to focus around ways to reduce the size of the rule list.

The Iterated Strip Rule Solver [1] identifies a class of two-dimensional patterns, called strip-rule patterns, which can be solved by coloring an entire row or column at a time. On this type of pattern, they can find a solution that is guaranteed to be within a 2-factor of the optimal solver. On other patterns, they divide it into a series of strip-rule patterns. They then show that the number of such sub-patterns is not too large. This allows them to achieve

a $O(\min(n^{1/3}, Opt^{1/2}))$-approximation ratio on range-ACLs and a $O(w^2 \min(n^{1/3}, Opt^{1/2})$-approximation ratio on prefix-ACLs.

Suri *et al.* created an optimal one-dimensional prefix compressor [27]. Their method starts with some base prefixes and determines which colors work best for their parent prefixes. For a monochromatic prefix, the best color is the given color. Otherwise, if the two child prefixes contain any of the same best colors, the parent contains those colors as its best colors; the children can put off using that color by letting the parent take care of it. Otherwise, its best set is their union; one child or the other will need a new rule and the other lets the parent take care of it. Once the root prefix is reached, one of the best colors is chosen arbitrarily. The rest of the tree is then climbed, placing a rule for any child whose best color set does not contain the color used for its parent. A generalized version that keeps track of the actual color costs allows for weighted colors.

Firewall Compressor [15] and TCAM Razor [14] observe that the one-dimensional problem can be solved in polynomial time for either range or prefix classifiers. They define a decision tree for the classifier with each node representing a field, edges the different values that the field can take, and leaves being rule lists for partial solutions. They work up the tree, solving a one-dimensional problem at the lowest nodes and replacing it with the resulting rule lists. TCAM Razor uses Suri's prefix solver from [27] while Firewall Compressor puts forth their own range solver. The two are later combined into ACL Compressor which can handle mixed prefix-range ACLs [16]. The primary limitation of this method is that similar, but not identical, sub-solutions must both be paid in full even if only a single point differentiates the two.

Topological Transform presents two transformation methods: Domain Compression and Prefix Alignment [18]. Domain Compression re-encodes a rule by mapping several equivalent

values to the same number. This reduces the number of bits required to encode a rule since there are fewer values. Prefix Alignment manipulates the encoded values further so that the ranges actually used by the rules tend to match up with prefixes. This reduces the cost of converting range rules to prefix rules.

2.2 Decision Tree

Decision tree methods partition the rules into multiple subsets by some method without having to search all of the rules. This method is then repeated on each part until the number of rules becomes small enough. Most methods then fall back onto a sequential search on the few remaining rules.

Most of these methods operate by cutting the domain into multiple regions. Since region boundaries may not match rule boundaries, this may require some rules to be replicated into multiple parts which increases memory usage.

HiCuts selects a field and divides the domain of that field into multiple equal sized regions [5]. Each rule is placed in whichever parts that overlap with their domain. In the worst case, a rule may ignore the field being cut. Such a rule must be replicated into every part. As such, HiCuts tends to have high memory usage. Once the number of rules in a part falls below a predetermined threshold, further partitioning stops. The remaining rules are then searched sequentially.

HyperCuts reduces the tree height by allowing multiple fields to be cut at once [22]. This allows for faster searching since HyperCuts is essentially doing two cuts at once. However, this does not reduce the amount of rule replication. As such, HyperCuts also tends to have large memory requirements.

EffiCuts reduces the amount of rule replication by partititioning rules into sets that work better together [31]. The rules are classified as being either specified or not specified in each field. Rules that are specified in the same fields end up in the same part. Each part is then made into its own tree. This dramatically reduces rule replication since no rule ever has to be copied into every child region (as some rules do with HiCuts and HyperCuts). However, the multiple trees tend to have a much larger total cost than HyperCuts; each tree is smaller but their combined cost is greater. They selectively merge some of the parts back together to help reduce this problem without reintroducing too much replication.

Rather than making many cuts, HyperSplit takes an alternate approach by making a single cut at a time that partitions the rules as effectively as possible [20]. While HiCuts and the rest divided the domain into equal-sized partitions, HyperSplit allows the two parts two be very different in size as long as they contain similar numbers of rules. As such, HyperSplit tends to have lower memory requirements than HyperCuts (but not as low as EffiCuts) at a relatively minor cost in tree height.

SmartSplit [7] attempts to combine HyperCuts and HyperSplit. They propose several heuristics for estimating the search time and memory usage of both HyperCuts and HyperSplit. This allows them to predict which method will produce a more efficient tree. Additionally, if they determine that a single tree will require too much memory, they can divide the rules into separate trees, like EffiCuts does. In SmartSplit, the rules are divided into four sets (unlike the 32 sets that EffiCuts uses) based on the specificity of the address fields. The set that is specified on both addresses is then recombined with the larger of the two single-address sets. This allows them to remove the primary source of rule replication (since all of the rules in the two larger sets have at least one field in common) without creating the extraneous trees that EffiCuts does.

In [26], the authors take a very different strategy. For a given field, they find the largest number of rules that are non-intersecting in that field. This allows them to order those rules by sorting them on that field. They repeat this process until all of the rules have been grouped. They then binary search each group until a matching interval is found. Afterwards, only a small number of rules must be searched to find a match. The chief downsides are that the search time is proportional to the number of independent sets required and that certain large rules may not be able to be placed into the same set as any other rule. We present work in Section 6 that improves on this method by significantly reducing the number of independent sets required.

In Grid-of-Tries [25], the authors consider using a multi-tiered set of tries where one field (such as source address) is encoded in a trie which stores pointers to another trie encoding a second field (such as destination address). To prevent rule replication, each rule is stored in only one of the tries and a switch pointer is used to jump from the trie associated with a more specific address to that of a less specific one without having to start back at the top of the second trie. This allows for $O(w)$ search times instead of $O(w^2)$.

In SAX-PAC [9], the authors propose dividing the rule list into separate lists that are order-independent. If a rule list is order-independent, then adding extra fields does not affect the overall search times as they can be ignored without affecting the ability to discriminate between rules. They invert this, showing how to remove fields while maintaining order-independence. They observe that for some existing solutions, such as Grid-of-Tries, an order-independent set can be searched in $O(\log^{d-1} n)$ time while using only linear memory. Thus, if the order-independent set can be reduced to only 2 dimensions, $O(\log n)$ search times can be achieved. To help keep the number of tables down, they recommend using a TCAM to solve the last 5-10% of rules. Because this is only a small fraction of the total

rules, it should be small enough to fit in the limited TCAM space.

2.3 Decomposition

There are many ways to efficiently search a single field, such as binary range search and longest prefix match. Methods in this category leverage this by breaking the problem into a series of one-dimensional problems and then aggregating the results. This first stage can be done in parallel, which allows these methods to be generally faster. However, the challenge lies in the second stage, where the results are aggregated together. This usually is memory inefficient, limiting the capacity of these methods.

In Parallel Bit-Vector (PBV) [11], the authors divide each field into intervals in the same manner as the effective grid discussed in Chapter 3. They then determine which rules intersect with each interval. They create a bit-vector for each interval where each bit corresponds to one of the rules. After finding a bit-vector for each field, they take the bitwise AND of all of the vectors; any bit set in the result corresponds to a matching rule. The most significant set bit will correspond to the first matching rule. The chief downside is that there can be $O(n)$ vectors that are each $O(n)$ bits long giving a rather unfavorable $O(n^2)$ memory requirement.

In Aggregate Bit-Vector (ABV) [2], the authors note that most rules do not intersect with most of the intervals. They group the rules into blocks and represent each block as a bit in a second bit-vector. When searching, they combine the aggregate vector first and then only combine and check the blocks corresponding to bits in the combined aggregate vector. Since they don't have to allocate memory for empty blocks, this reduces overall memory usage (although not asymptotically).

The authors of Cross-Producting [25] note that for most fields, the number of unique values is significantly less than the total number of rules. They construct a list of best-matching values for each field. They then construct a table by taking the cross product of these lists, mapping each tuple to the corresponding result. They store the result in a hash table. This method can provide very high throughput, but the size of the hash table can be up to $O(n^d)$, which is generally too big. They introduce a variant called On-Demand Cross-Producting that limits the table size and treats it more like a cache, but then performance is largely dependent on the behavior of the packet stream.

2.4 Tuple Space

In this category, rules are grouped together by the number of specified bits in each field, which is called a tuple. In the Tuple Space Search classifier [24], all rules with the same characteristic tuple are placed into a hash table. During a search, they combine the tuple with a packet to form a hash key, allowing the table to be probed. Since each rule can only be placed in one table, and hash tables have amortized constant-time updates, this method has recently had a resurgence with the Open vSwitch OpenFlow framework where rule updates happen frequently [19].

Nonetheless, Tuple Space Search is hampered by the number of tables required; each unique characteristic tuple requires its own separate table; potentially w^d tables could be required. Several solutions are proposed to the problem, each with their own drawbacks.

One solution is to only use certain fields (usually source and destination addresses), which limits the exponential factor [24]. Some rules may be identical on the included fields but different on the excluded fields. These rules will collide and necessitate a linear search to

determine the true match. In some cases, a poor field choice may exclude a particularly discriminatory field causing a prohibitively large search to be required.

An alternate solution is to use one or two one-dimensional classifiers (similar to the decomposition classifiers) as a pre-classifier to prune out tables that do not contain any matching rules [24]. These pre-classifiers usually are more difficult to update and maintain, increasing update times. This unfortunately removes the primary advantage that Tuple Space Search has over other classifiers.

Chapter 3

Terminology

Here we present definitions and terminology used in common for all of the remaining sections.

$\mathcal{K}$ is the set of all decisions such as {Accept, Deny}, {Black, White}, or {1, 2, 3, 4}. A color $c \in \mathcal{K}$ is any one of those decisions.

An interval $s = [l, u]$ is the range of integers between l and u, inclusive. A prefix $s = \{0, 1\}^k\{*\}^{w-k} = p\{*\}^{w-k}$ is a special type of interval that covers the range $[p \cdot 2^{w-k}, p \cdot 2^{w-k} + 2^{w-k} - 1]$. More idiomatically, $*^w = [0, 2^w - 1]$ and replacing the first $*$ of s with a 0 reduces the range to the lower half and with a 1 to the the upper half.

The domain $\mathbb{C}$ is Cartesian product of d intervals (normally $[0, 2^{w_i} - 1]^d$). A box B is a subset of $\mathbb{C}$ that is also the Cartesian product of d intervals.

A pattern is a function $P : \mathbb{C} \to \mathcal{K}$. P thus assigns a color to every point in $\mathbb{C}$. For any point $x \in \mathbb{C}$, $P(x)$ is the color of that point. Two patterns P_1 and P_2 are equivalent if $P_1(x) = P_2(x)\ \forall x \in \mathbb{C}$.

A rule $r = (B, c)$ assigns $B \subseteq \mathbb{C}$ the color $c \in \mathcal{K}$. A rule list $L = [r_1, r_2, r_3, \cdots, r_n]$ is an ordered list of n rules. A rule list defines a pattern with $L(x) = c_i$ if $x \in B_i$ and $x \notin B_j\ \forall j < i$. More simply, the color $L(x)$ is the color of the first rule that matches x. We abuse notation slightly by using L to refer to both the rule list and the pattern it generates. Two rule lists are equivalent if they produce equivalent patterns.

A box is monochromatic if $\exists c$ such that $P(x) = c\ \forall x \in B$. We can partition $\mathbb{C}$ into

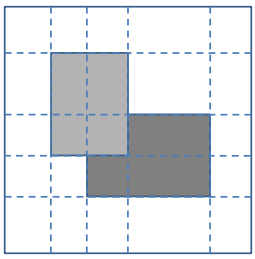

Figure 3.1: An effective grid denoted by the dashed lines

monochromatic boxes by using the concept of an effective grid from [1]. The effective grid is the set of boundary lines that separate two differently colored regions in P. It has sizes $m_1 \times m_2 \times \cdots \times m_d$ with each $m_i \leq 2n - 1$. Each cell in the effective grid will be a monochromatic box. We can use this property to create a transformed problem where each point in the transformed problem represents an entire effective grid cell in the original problem. This reduces the domain space that must be considered.

Chapter 4

Diplomat

Generally speaking, if two rule lists are equivalent, the shorter rule list is preferable to the longer one. When doing a basic sequential search, it generally takes less time to search a smaller list than the longer one. Additionally, some devices use TCAMs to search the entire lists in parallel. Since TCAMs are expensive in both upfront costs and power, a rule list that can fit into a smaller TCAM can achieve significant savings.

Thus, given a rule list L, the objective is to find the smallest possible equivalent rule list L'; that is to find the rule list with the fewest number of rules that has the same decision for every possible input packet. This was shown to be NP-Hard in [1] for $d \geq 2$ for the range version. The prefix version is also believed to be hard. The one-dimensional versions of both problems are in P.

In this chapter, we review the prior ACL Compressor [16]. We then review our Diplomat work which corrects a key limitation of ACL Compressor. This chapter is based on our previous published work [4].

4.1 ACL Compressor

ACL Compressor [16] is a combination of Firewall Compressor [15] and TCAM Razor [14]. Firewall Compressor compresses range rule lists while TCAM Razor operates on prefix rule lists. However, the basic mechanism behind the two is the same, allowing ACL Compressor

to operate on mixed rule lists.

ACL Compressor begins by creating a Firewall Decision Diagram [13] for the rule list. In an FDD, each of the leaf nodes represents a possible decision, such as Accept or Deny. Each of the branch nodes represents a field, such as source IP address or protocol. Edges between nodes represent ranges of values that a given field can take.

Each of the lowest branch nodes represents a one-dimensional problem. They solve this with the Firewall Compressor 1-D solver if it is a range field or Suri's 1-D solver [27] if it is a prefix solver, and replace the branch node with the resulting rule list. Both methods are covered in Section 4.3.

Higher branches will eventually find that their children have been replaced with rule lists. They then become weighted one-dimensional problems with each rule list being a color with weight equal to the number of rules in the list. Fortunately, the weighted problem is still in P and can be solved with modified versions of the above algorithms.

The downside to this method is that it only knows if two rule lists are the same or not. If two subproblems are similar, say because their patterns differ at only a single point, it must still pay for both rule lists in their entirety. ACL Compressor runs redundancy removal [12] to remove much of this, but not all of it can be removed this way.

4.2 Diplomat Overview

Each of the lower nodes in the FDD represents a smaller dimensional subproblem. The chief handicap of ACL Compressor is that it considers each of these subproblems to be independent. However, neighboring subproblems usually are similar to one another. Our method, Diplomat, overcomes this limitation by considering neighboring subproblems together.

Instead of solving each subproblem separately, Diplomat compares neighboring subproblems and determines where the patterns are different. Diplomat then defines those differences using the minimum number of rules. This normally requires fewer rules than completely defining the entire subproblem. Afterwards, the two subproblems can be considered to be identical and merged together as any differences are defined by the previously placed rules. Diplomat repeats this process until only one subproblem remains. It then solves this subproblem fully.

Diplomat requires three parts. One of these parts is how to solve the final subproblem. The second is how to merge two subproblems together by resolving the differences between them. The final part is how to determine which order to merge the subproblems.

4.3 Subproblem Solver

Normally, we solve a subproblem by making a recursive call to Diplomat. This subproblem has one fewer field than the original problem and so eventually will be reduced down to a single field. This one-dimensional problem is then handled differently. If it is a range field, we use the one-dimensional solver from Firewall Compressor [15]. If it is a prefix field, we use the one by Suri *et al.* [27]. We describe both methods here.

4.3.1 Firewall Compressor

Given some input classifier L, compute the effective grid of L and label the cells from 0 to $m-1$. The first key insight is that the last rule r_n covers the leftmost cell 0 in some optimal solution. Liu *et al.* use dynamic programming to determine which other cells are also covered by rule r_n. If no other cells are covered, then the optimal solution consists of rule r_n which

only covers cell 0 plus an optimal solution that covers region $[1, m-1]$. Otherwise, let x be the nearest cell that is also covered by r_n. The optimal solution is then one that covers region $[1, x-1]$ and one that covers region $[x, m-1]$ where the last rule covers cell x (and cell 0). We use this method for constructing our range resolvers.

4.3.2 Suri's Algorithm

Their method works by dividing the range into regions designated by prefixes. Each region has a designated background color. They then proceed to merge regions together, sharing background colors between neighboring regions at reduced cost when possible.

More formally, they begin by dividing P into the largest prefix segments that are monochromatic. For each prefix-color pair, compute $Cost(p, c)$, where c is the background color applied to the given prefix. For the base case of one of the monochromatic prefixes, then $Cost(p, c) = 0$ if c is the color of the segment and 1 otherwise. For other prefixes, $Cost(p, c) = \min_{c_0,c_1}(Cost(p0, c_0) + Cost(p1, c1) + [c \neq c_0] + [c \neq c_1]$. The final cost, $Cost(p) = \min_c Cost(p, c) + 1$ (to fill in the background color). We utilize Suri's algorithm for constructing our prefix resolvers.

4.4 Resolvers

The second part to Diplomat is the resolver that is capable of finding and settling the differences between two patterns. The resolver is given two subproblems, P_t and P_b over the same fields and returns a rule list ℓ. In the one-dimensional case, we refer to P_t and P_b as rows. Otherwise, we call them planes.

The resolvers presented here assume that the resulting rules must come from P_t allowing

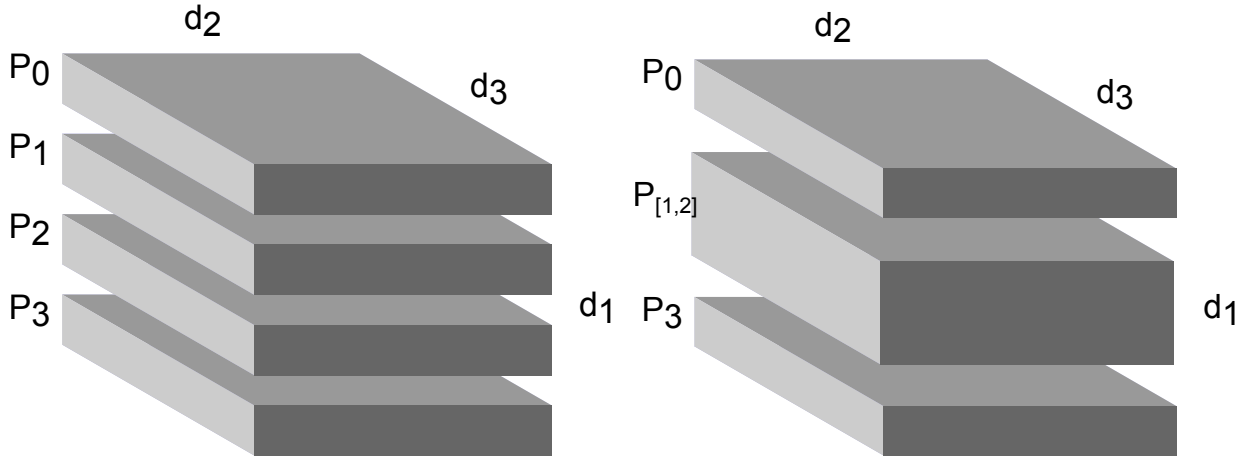

Figure 4.1: Merging planes P_1 and P_2.

P_b to be used as the merged subproblem. With this restriction, for any point $x \in \mathbb{C}$, either $P_t(x) = \ell(x)$ or $P_t(x) = P_b(x)$ and $\ell(x)$ is undefined. In the original paper [4], we also presented alternate versions that allowed rules to come from either subproblem. However, this restriction allows the use of the dynamic programming scheduler in Section 4.5.

4.4.1 One-Dimensional Range Resolver

Here, we present an optimal way to merge P_t and P_b between columns 0 and k. Let ℓ_k denote the partial rule list returned by this optimal solution and let $C_k = |\ell_k|$. If $P_t(k) = P_b(k)$, then $\ell_k = \ell_{k-1}$, and $C_k = C_{k-1}$ as there are no differences in column k.

Otherwise, there is a difference between P_t and P_b in position k. This means ℓ_k must include at least one rule containing k of color $P_t(k)$. Call this rule r_m. Similar to Firewall Compressor, we can assume that r_m is the last rule in ℓ_k. Let $B_m = [s, k]$ for some point s in $[0, k]$. Since ℓ_k is defined over all of $[s, k]$, $\ell(x)$ must equal $R_t(x)$ for all x in $[s, k]$. The cheapest way to do that is by using Firewall Compressor. We also need to resolve the range $[0, s-1]$. This can be done with another application of this method.

This allows us to set up a dynamic program. For each k, $C_k = \min_{0 \le s \le k}(C_{s-1} + |FC(P_t[s, k])|)$

and $\ell_k = \ell_{s-1} \cdot FC(P_t[s,k])$ for the minimum s. As a base case, $C_{-1} = 0$ and $\ell_{-1} = \emptyset$.

4.4.2 One-Dimensional Prefix Resolver

Here, we present an optimal way to merge P_t and P_b over the range of some prefix p_r using only prefix rules. We begin by observing that $Suri(P_t)$ is a resolver. Let p denote the smallest prefix (and thus specifies the largest number of points) used in an optimal resolver. The restrictions on prefix rules means that either $p = p_r$, $p \subseteq p_r0$, or $p \subseteq p_r1$. If $p = p_r$, then the optimal resolver is $Suri(P_t)$. Otherwise, it is the combination of the optimal resolvers for p_r0 and p_r1. Let ℓ_p be the optimal resolver for P_t and P_b over the range denoted by p. Then, $\ell_{p_r} = \min(\ell_{p_r0} \cdot \ell_{p_r1}, Suri(P_t))$. Our base case for this recurrence relation occurs when $P_t(p_r)$ is monochromatic meaning $|OptResolve|$ is 0 if $P_t(p_r) = P_b(p_r)$ and 1 otherwise.

4.4.3 Higher-Dimensional Resolver

If $d \geq 3$, we need to merge two patterns P_t and P_b that are $(d-1)$-dimensional hyperplanes. This is more complicated than merging two rows. In particular, we have an optimal algorithm for compressing 1-dimensional patterns, but compressing 2-dimensional patterns is NP-hard as shown in [1]. This difference in complexity carries over to merging patterns.

We propose two different in-plane methods for merging $(d-1)$-dimensional hyperplanes when $d \geq 3$. Without loss of generality, we assume that we wish to merge P_t and P_b leaving pattern P_b. Thus, all rules in the resolver will be within P_t.

The first method is to enumerate all the differences. That is, we use the FDD comparison algorithm in [13] to find the differences between the two patterns. We then define ℓ_{dif} by generating rules in P_t to cover all the places where P_t differs from P_b. We may cover

additional locations if this reduces the total number of rules required to cover all the differences. This resolver implicitly uses the correct type of rule (range, prefix or mixed) given the characteristics of the fields.

In the second method, we find *box*, the minimum bounding box surrounding all differences. If any of the fields are prefix fields, we expand the bounding box in that dimension so that the bounding box can be defined by prefixes. We then solve $P_t(box)$ using the appropriate version of Diplomat (range, prefix or mixed) and call the resulting solution ℓ_{mbb}.

To achieve greater compression, we run both methods and use the smaller of the two resolvers. That is, our final resolver $\ell = \min(\ell_{dif}, \ell_{mbb})$.

4.5 Schedulers

The final part of Diplomat is a scheduler which determines the order that planes are to be merged together. In general, different schedules will give different results.

In this section we present two schedulers: one for range rule and one for prefix rules. Both use dynamic programming to consider all possible schedules. This requires the resulting rule list of each merge to always be one of the original planes; otherwise we would need to store all of the partial solutions. For this reason, our resolvers all draw rules from only one plane. On a mixed rule list, Diplomat picks a scheduler appropriate to the field being split on.

4.5.1 Range Scheduler

We begin by precomputing the cost of each possible merge as $MergeCost[i,j] = \left|Resolve(P_i, P_j)\right|$ $\forall i \neq j \in [0, m_1 - 1]$ where the returned pattern is P_j. This requires $O(m_1^2)$ calls to *Resolve*. However, because the merged pattern is always one of the original patterns, we do not need

to do any more merges.

We then define the following subproblem. Given interval $[l, u]$ for $0 \leq l \leq u \leq m_1 - 1$, we find the minimum cost for merging patterns $P_l, \ldots P_u$ together while having P_r be the remaining pattern where $l \leq r \leq u$. We use standard dynamic programming techniques to compute the minimum cost solution to these $O(m_1^3)$ subproblems and the corresponding optimal schedule of merges.

4.5.2 Prefix Scheduler

We also present a dynamic programming method for scheduling that works with prefixes. We begin by precomputing the merge costs as for the range scheduler above. We then define the recursive subproblem. Given a prefix p and row $r \in p$, we find the minimum cost for merging the two halves, $p0$ and $p1$ while having P_r be the remaining pattern. If $r \in p0$ then $Cost(p, r) = Cost(p0, r) + \min_s(Cost(p1, s) + MergeCost[s, r])$. The case for $r \in p1$ is similar. As a base case, $Cost(p, r) = 0$ if $p = r$. We use standard dynamic programming techniques to compute the minimum cost solution to each of the $O(w^2 m_1^2)$ subproblems and the corresponding schedule of merges.

4.6 Results

In this section we evaluate the effectiveness of Diplomat on real-life classifiers. Specifically, we assess how much Diplomat improves over ACL compressor [16], the current state of the art ACL compression algorithm. We test range, prefix, and mixed ACL compression. For mixed-ACLs, we consider the port fields as ranges and the IP and protocol fields as prefixes.

4.6.1 Methodology

Diplomat and ACL Compressor (AC) are both sensitive to the ordering of the five packet fields (source IP address, destination IP address, source port, destination port, and protocol). We run both algorithms using each of the $5! = 120$ different permutations across all of the fields and use the best of the 120 results for each classifier. We also use redundancy removal [12] as a post-processing step.

We now define the metrics for measuring the effectiveness of our Diplomat variants. Let f and g denote compressors and L denote a classifier. Let $f(L)$ denote the resulting rule list obtained by applying compressor f to L, and $|L|$ is the number of rules in L. The *compression ratio* of f on L is $\frac{|f(L)|}{|L|}$, and the improvement ratio of f over g on L is $\frac{|g(L)|-|f(L)|}{|g(L)|}$. In our results, we focus on the mean compression ratio and mean improvement ratio given a set of classifiers S. It is desirable to have a low compression ratio and a high improvement ratio.

We assess the performance of our algorithms using a set of 40 real-life classifiers which we first preprocess by removing redundant rules. We divide this set into three smaller sets based on the number of rules after running redundancy removal. The small set contains the 17 smallest classifiers, the middle set contains the next 12 larger classifiers, and the large set contains the 11 largest classifiers. The classifiers in the large set all have at least 600 rules after redundancy removal, with the largest having more than 7600 rules.

In general, larger classifiers also have larger effective grids. Not all of the fields strictly increase as the list size goes up. In particular, the protocol field only uses a few distinct values across all classifiers, usually 6 (TCP) and 17 (UDP). As such, the smallest dimension size is at most 9 for all classifiers. The range of the largest dimension size for each set can be seen in Table 4.1.

	Min	Median	Max
Small	5	14	37
Medium	24	66	139
Large	55	506	1805

Table 4.1: Size of Largest Dimension

4.6.2 Compression Results

Our results show that (i) Diplomat outperforms ACL Compressor and that (ii) the performance gap between Diplomat and ACL Compressor increases as the classifiers grow in size and complexity. Diplomat's superior performance manifests itself in two ways. First, as classifiers grow in size and complexity from small to medium to large, the percentage of classifiers where Diplomat outperforms ACL Compressor increases from almost 0% for the small classifiers to over 50% for the medium classifiers to roughly 100% for the large classifiers (Figure 4.3). Second, again as classifiers grow in size and complexity, Diplomat's average improvement ratio grows from almost 0% for the small classifiers to roughly 10% for the medium classifiers to as high as 30% for the large classifiers (Figure 4.2).

It is especially encouraging that Diplomat performs best on the large classifier set as the large classifiers are the ones that most accurately represent the complex classifiers we are likely to encounter in the future and which most need new compression techniques. More specifically, we hypothesize that for the small classifiers, there is relatively little room for optimization and both methods are finding optimal or near optimal classifiers. However, as the classifiers increase in size and complexity, Diplomat is able to find new compression opportunities missed by the earlier methods. More detailed compression ratio results for Diplomat on range, prefix, and mixed-ACLs are shown in Figures 4.4, 4.5, and 4.6, respectively.

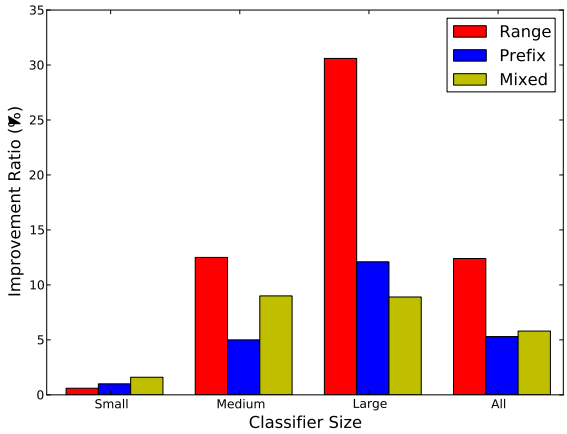

Figure 4.2: Mean improvement ratios

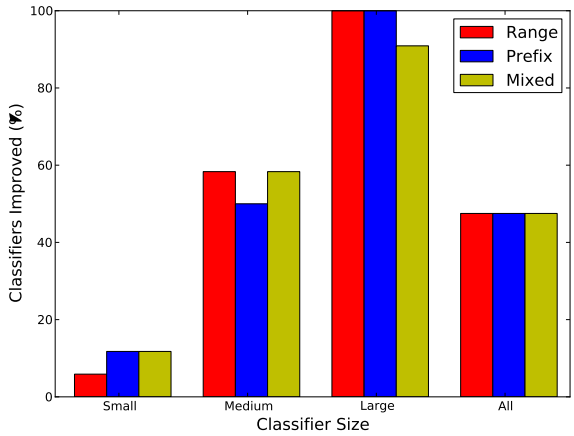

Figure 4.3: Percentages of classifiers Diplomat outperforms previous methods

4.6.3 Efficiency

We implemented our algorithms using a combination of C# and VB. We found that Diplomat runs in under a second on the smaller sets, between a second to a minute on the medium sets, and up to a few hours for some permutations on the largest sets. Our implementation has not been designed with an emphasis on speed. In particular, multiple calls to Firewall Compressor or Suri are made for each row when compared to any other row. Implementing some optimizations such as remembering this result should result in significant speed ups. Furthermore, since Diplomat is additive, we can terminate permutations once they have

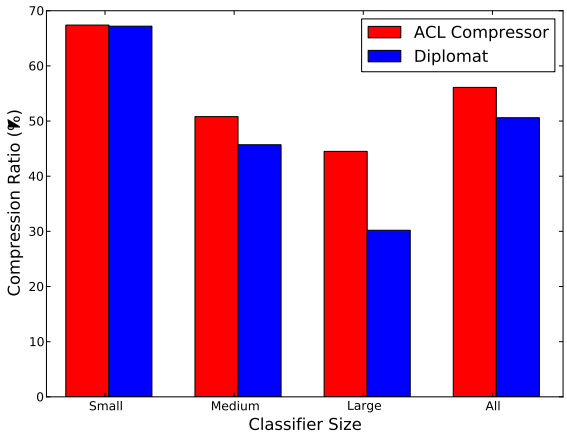

Figure 4.4: Compression ratios on range classifiers

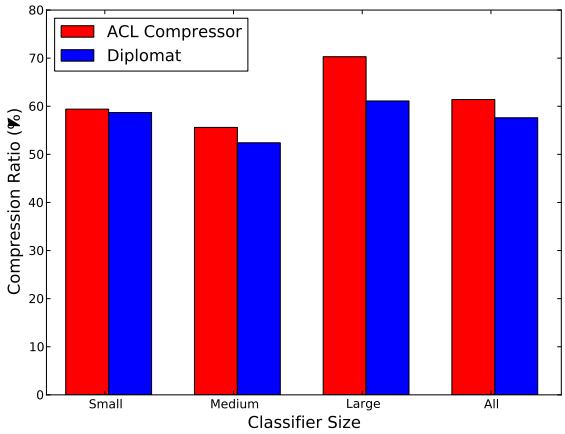

Figure 4.5: Compression ratios on prefix classifiers

accumulated more rules than a known solution such as one generated by another permutation. This allows us to safely reduce the amount of work that must be done. Additionally, since better permutatations tend to complete relatively quickly, we can terminate permutations that run for a long time with little likelihood of missing the best solution.

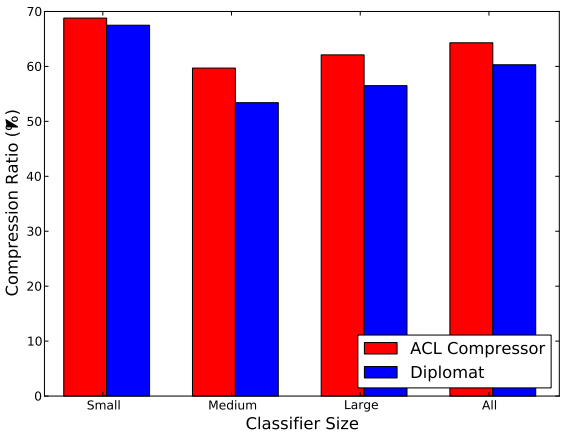

Figure 4.6: Compression ratios on mixed classifiers

Chapter 5

ByteCuts

In Chapter 4 we devised a way to minimize the size of a rule list. This is especially helpful for supporting hardware-based packet classifiers, such as TCAMs, which have limited space available. Unfortunately, the size of rule lists has been growing faster than can be supported by TCAMs, even with compression.

In this chapter, we will instead develop a software-based packet classifier which we call ByteCuts. ByteCuts builds search trees similar to several existing methods such as Hypercuts [22] and HyperSplit [20]. ByteCuts adds two new improvements over existing search tree classifiers. First, it uses an intelligent rule separation scheme that minimizes the rule replication problem that plagues most of these tree-based classifiers. Second, it introduces a new cutting method that reduces the height of the search trees.

Rule	Source	Dest
r_1	000	101
r_2	001	101
r_3	010	001
r_4	011	001
r_5	011	000
r_6	100	***
r_7	110	***
r_8	***	011
r_9	***	010

Table 5.1: A sample rule list

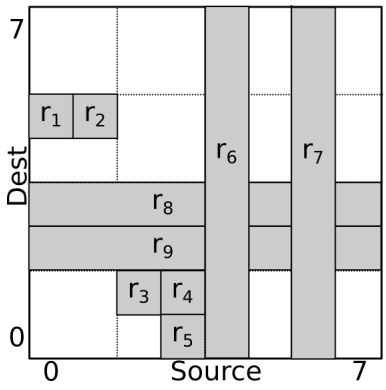

Figure 5.1: A graphical representation of the rules from Table 5.1.

5.1 Related Work

5.1.1 HiCuts

HiCuts [5] is one of the earliest decision tree based methods. In HiCuts, one of the fields is selected and the domain of that field is split into several equal-sized pieces. Each rule is then copied into the sublist corresponding to whichever subdomain it is in. Unfortunately, the domain boundaries and the rule boundaries do not always match up. In this case, the rule must be copied into the sublist of each domain that partially contains it. In the worst case scenerio the rule completely ignores the selected field and must be copied into every sublist. This is the rule replication problem and it causes drastically increased memory sizes, even if only a pointer to the rule is copied.

This process is repeated on each of the sublists. Eventually, the number of the rules in the list becomes small enough that further splitting is of little benefit. Thus, when the number of rules becomes less than a specified threshold, it switches over to a sequential search instead.

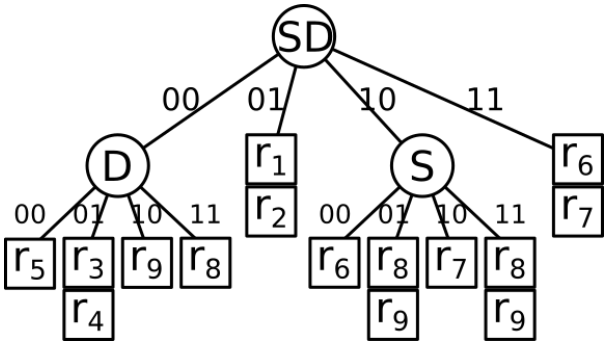

Figure 5.2: The tree HyperCuts builds for Table 5.1.

5.1.2 HyperCuts

HyperCuts [22] improves on HyperCuts by allowing multiple fields to be cut at once. This reduces the overall tree height by combining multiple levels into one decision. Unfortunately, this does not reduce the amount of rule replication. They attempt to alleviate replication by allowing branch nodes to also have rule lists, but since few rules meet this criterion (as multiple fields are cut), this is not as helpful as it could be. A sample HyperCuts tree can be seen in Figure 5.2.

5.1.3 EffiCuts

EffiCuts [31] is a variant on HyperCuts that addresses the rule replication problem directly. For each rule they assign it a tuple corresponding to whether it uses a particular field or not. Then then group all of the rules that have the same tuple together. Then then construct a HyperCuts tree for each group. Since every rule in the group is specified in whichever field is cut, only a few rules need to be replicated into some of the parts. While each tree is also smaller than the single HyperCuts tree, their sum cost tends to be significantly higher. They combat this by selectively merging groups that differ in only one field together before creating the tree. This leaves enough options for field choices while reducing the number of

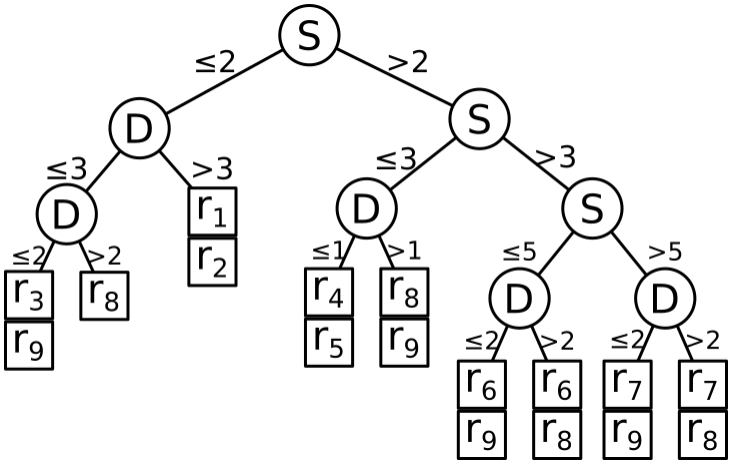

Figure 5.3: The tree HyperSplit builds for Table 5.1.

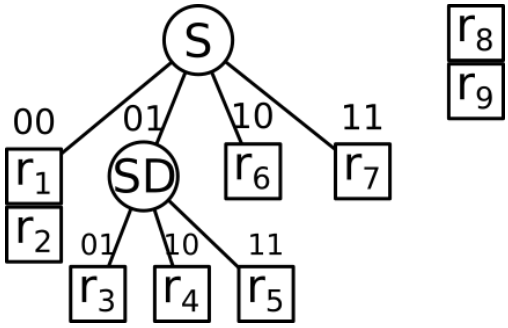

Figure 5.4: The tree SmartSplit builds for Table 5.1.

trees required.

5.1.4 HyperSplit

HyperSplit [20] takes a different option by making a single cut in an arbitrary location as opposed to the HiCuts family's many equal-sized cuts. They first choose the field that has the most even distribution of rules across the domain. They then make a cut that balances rule separation and the amount of overlaps. This method tends to have significantly less rule replication than HyperCuts (although more than EffiCuts) as it selects cuts that divide only a few rules and those that are divided are replicated into only two subtrees. A sample HyperSplit tree can be seen in Figure 5.3.

5.1.5 SmartSplit

SmartSplit [7] improves on HyperCuts and HyperSplit in two ways. First, they describe a method for estimating the search times and memory requirements for HyperSplit and HyperCuts. This allows them to predict which algorithm would be better given their needs and constraints. Second, they modify the EffiCuts separation scheme by using only the two address fields and only if they estimate that there will be too much rule replication without it. This method produces four subsets (specified on source address, specified on destination address, specified on both, and specified on neither). The specified on both set is then merged with one of the partially specified sets. This leaves three sets that generally have sufficient cohesion to produce a tree without significant rule replication. A sample SmartSplit tree can be seen in Figure 5.4.

5.2 Methodology

In this section, we describe our ByteCuts (BC) classifier. ByteCuts improves on existing tree-based packet classifiers in three ways. First, it incorporates a new cutting method that is more versatile than the one included in HiCuts [5] and HyperCuts [22]. Second, it includes a tree selection algorithm that is more discriminating than SmartSplit [7]. Finally, it can include both cut nodes and split nodes (like HyperSplit [20]) in the same tree, unlike SmartSplit where each tree is formed entirely of one kind of node (although different trees can be of different types).

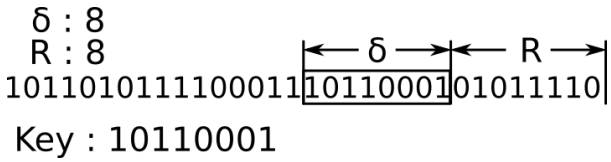

Figure 5.5: 8 bits are extracted from this packet to form a key.

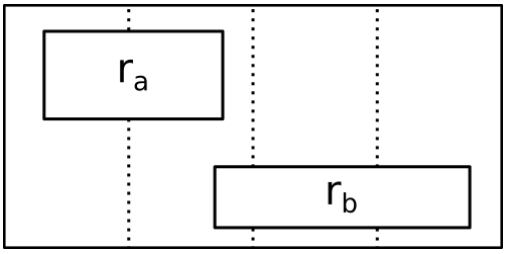

Figure 5.6: Rule r_a is replicated twice and r_b is replicated three times.

5.2.1 Bit Cutting

ByteCuts works by extracting k consecutive bits from a rule or packet, interpreting those bits as a number, and using that number as an index into an array of child nodes. This strategy is designed to work on ternary rules where neighboring bits are likely to appear together. An example of this process can be seen in Figure 5.5.

Address fields, the most common ternary field, are usually represented by prefixes, and due to how addresses are usually allocated, certain prefix lengths are much more common then others. We leverage this to reduce the number of cases to consider. We will only consider *nybbles*, which are blocks of 4 bits (half a byte). There are 8 nybbles per address field. For simplicity, our examples use only a few bits. This precludes demonstrating the nybble alignment.

Port fields are normally not represented using prefixes or other ternary expressions but are instead represented using ranges. We do not use our cutting method on these fields. Instead, we borrow the splitting method from HyperSplit [20] on these fields.

Rule	Source	Dest
r_1	000	101
r_2	001	101
r_3	010	001
r_4	011	001
r_5	011	000
r_6	100	***
r_7	110	***
r_8	***	011
r_9	***	010

Rule	Source	Dest
r_5	011	000
r_3	010	001
r_4	011	001
r_9	***	010
r_8	***	011
r_1	000	101
r_2	001	101
r_6	100	***
r_7	110	***

Table 5.2: Both (Source, 3) and (Dest, 3) create trees with a maximum collision limit of 2 and omit 2 rules.

5.2.2 Tree Selection

Tree based methods, including ByteCuts, suffer from a problem called *rule replication*. Rule replication is caused by rules falling into multiple partitions due to wildcards which forces them to be copied multiple times. This can be seen in Figure 5.6 where rules r_a and r_b are both replicated into multiple partitions. Existing methods, such as EffiCuts [31] and SmartSplit [7] have attempted to alleviate this problem by partitioning rules such that all rules in the same partition have similar characteristics. This significantly reduces the amount of replication required as it is easier to find good cuts that do not trigger any of the wildcards.

We also divide the rule list L into a list of trees $\mathcal{T}$ based on the characteristics of the rules. We consider each possible length for a prefix field (in nybble increments); this gives 8 possible lengths for each address field. For a particular field-length pair, we divide the rules into two sets: those that are sufficiently long and those that are not. For each rule that is sufficiently long, we find its prefix of the given length. We find the maximum occurrence of any of the prefixes. This approximates the largest partition of a single cut; smaller maximums are preferable. The other set represents rules that will be excluded from the tree; smaller numbers here are also preferable.

These two goals are at odds with each other. Decreasing the length excludes fewer rules but increases the maximum partition size. We deal with this trade-off by striking a balance between the number of excluded rules and the maximum partition size.

We define two thresholds: one for each objective. We define c to be the maximum collision ratio and $n_c = c \cdot n_L$, where n_L is the number of rules in L that have not yet been placed into a tree. Likewise, we define χ to be the maximum exclusion ratio and $n_\chi = \chi \cdot n_L$. Both n_c and n_χ will be smaller for later trees since n_L will be smaller; n_c defines what an acceptable collision limit is while n_χ defines the desired maximum number of excluded rules.

If any field-length pair (f, w_f) has a collision limit below n_c, then we select the pair that minimizes the number of excluded rules. Otherwise, from those that exclude at most n_χ rules, we select the pair that minimizes the collision limit. If neither objective is reachable, we select the pair that minimizes the sum collision size + rules excluded.

Once a field-length pair (f, w_f) is selected, we create a tree from all of the rules whose prefix length is at least w_f on field f as described in the next section. We then repeat the tree selection process on the remaining rules.

Once the number of remaining rules falls below a certain threshold, τ (5% in our experiments), we declare the remaining rules "bad" (and by extension, the previous rules "good"). Bad rules typically have relatively few bits specified in both address fields. If TCAM is available, we place the bad rules into TCAM. For a pure ByteCuts solution, we try to build a single tree with all the bad rules and only remove rules if they require too much rule replication in the cutting phase (see Section 5.2.3). Specifically, we only remove a rule from a tree because of rule replication if it has 5 or more wildcard bits within any individual cut in the tree.

Consider the rules in Table 5.1. Both (Source, 3) and (Dest, 3) have a maximum partition

size of 2 and 2 excluded rules, making them equally valid. Other options have a larger maximum partition size; they are not considered. Choosing (Source, 3) will create a tree from rules r_1 to r_7. Rules r_8 and r_9 will be turned into a separate tree.

When we construct each tree T_i, we associate with it the highest priority of the rules that it contains. We then sort $\mathcal{T}$ so that if $i < j$, then T_i has a higher priority than T_j. This will be helpful for performing packet classification. Specifically, if a high-priority rule is matched in an earlier tree, the trees containing only lower-priority rules do not need to be searched.

5.2.3 Cut Selection

Once the rules for a tree are selected, ByteCuts partitions the list into several sublists using the method described below. These sublists are then partitioned in the same manner. This process repeats until a partition has at most n_{leaf} rules. This leaf node will then be searched sequentially. In our experiments, we set $n_{leaf} = 8$.

Our primary means of partitioning is with *cuts* on ternary fields (including prefix and exact-match fields). For each cut, we select up to k consecutive bits. We extract those k bits from each rule and interpret it as a number. All rules that have the same number will end up in the same partition. If a rule has a wildcard in any of those bit positions, then the rule is replicated into several partitions.

For each field, we consider each possible left-most bit and select the next possible δ bits for each $0 < \delta \leq k$. This gives us a selection of δ bits. From each rule r_i, we then extract these bits and interpret it as a number. Rules with wildcards count for multiple numbers (by replacing the * with both 0 and 1). We count how many times each number appears. We select the cut that has the smallest maximum frequency. This cut will minimize the largest subtree. In case of ties, we prefer cuts with smaller δ since this saves memory.

Rule	Source	Dest	Key
r_1	000	101	00
r_6	100	***	00
r_2	001	101	01
r_3	010	001	10
r_7	110	***	10
r_4	011	001	11
r_5	011	000	11
r_8	***	011	
r_9	***	010	

Table 5.3: ByteCuts uses the two highlighted bits from the Source field as a key to cut the first tree. The second tree does not need cutting since it contains only two rules.

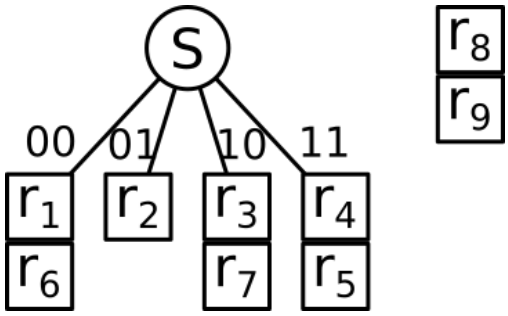

Figure 5.7: A graphical representation of the trees produced in Table 5.3.

Once a cut is selected, we partition the rules. We create 2^{δ} partitions. Similar to above, from each rule r_i, we extract the selected δ bits, interpret them as a number, and then place r_i into the partition corresponding to that number. Rules with wildcards will be placed into multiple partitions. We then recurse on each partition to create a new subtree. Some lists contain identical sets of rules (because of wildcards). We detect this and create only one child node that is shared for each of these sublists.

For example, consider the rules from Table 5.1. In Section 5.2.2, we selected rules r_1 to r_7. For this example, we let $n_{leaf} = 2$ and $k = 2$. If we select the first two bits in the Source field, then one partition will contains rules r_3, r_4, and r_4, but if we select the last two bits, then each partition has at most 2 rules making it a better choice. None of the cuts on the

Dest field are as good since they require replicating r_6 and r_7 into all partitions. We thus select (011, 000) as the mask for our cut. This yields four children, with part 00 containing r_1 and r_6, 01 containing r_2, 10 containing r_3 and r_7, and 11 containing r_4 and r_5. After this cut, each partition only has $2 \leq n_{leaf}$ rules, so we create a leaf node for each part.

We have two optimizations to make cut selection faster. First, if a rule has more than 4 wildcards in the selected bits we assume that it ends up in every partition. We add the number of such rules as a penalty rather than counting them towards individual partitions. This prevents having to increment thousands of frequencies for rules that have many wildcards.

The second optimization is that we align all of our cuts to nybble boundaries: our minimum cut size is 1 nybble, we increment the cut length by 1 nybble, and in our experiments our maximum cut length is $k = 4$ nybbles (16 bits). This limits the size of the child array to at most 65536 nodes; increasing this limit increases the array size, and thus potential memory usage, exponentially. We select only nybbles because, in practice, these lengths are significantly more common than other lengths. Thus finer cuts add few benefits at significant time costs.

5.2.4 Splitting

For range fields, we borrow the splitting method from HyperSplit [20]. In a split, a field f and split point p are selected. The rules are divided into two sublists; all rules where $low(r[f]) \leq p$ are placed into the first list and all rules where $high(r[f]) > p$ are placed into the second list (some large rules will end up in both lists, causing replication).

We currently use a selection heuristic not detailed in the HyperSplit paper, which we call rule-balanced. In the rule-balanced heuristic, the split point is selected so that the larger set contains as few rules as possible. The dimension with the lowest max size is selected as

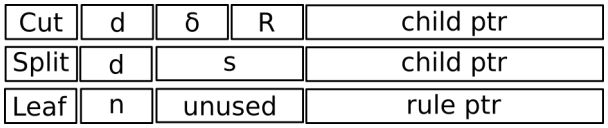

Figure 5.8: A depiction of the memory layout for each of the three types of nodes. Each includes a 4 byte header and a 4 byte pointer.

the best split. This works well with our bit cutting selection; in both cases we are trying to minimize the worst case size of any sublist.

Since we only allow rule splitting on port fields, we only need 16 bits to represent the splitting point whereas HyperSplit needs 32 bits. This allows us to use the remaining 16 bits to convey other information such as the split dimension and node type. This allows us to use 32 bits for the child pointer compared to HyperSplit's 24 bits, increasing the maximum rule capacity.

5.2.5 Node Design

ByteCuts uses nodes that are 64 bits in size. We use 2 bits to represent the type of node; leaf, cut, or split. We pad this to 8 bits to better align each field. In each case, 32 bits are used for an array pointer. The remaining 24 bits are used slightly differently between each of the node types. The layout for each type can be seen in Figure 5.8.

For the cut nodes, we use 8 bits each for the dimension, δ, and right endpoint. Split nodes use 8 bits for the dimension and 16 bits for the split boundary. Since we only split on the 16 bit range fields, we do not need the full 32 bits that HyperSplit used for the split boundary. Finally, for a leaf node we can use 8 bits to store the number of rules associated with this node (leaving 16 bits unused).

For the cut node we created in Section 5.2.3, we need to encode the mask (011, 000). This selects $\delta = 2$ bits from dimension 0. This uses the least significant bits, so the right

end point is 0. If we use 1 as the ordinal for a cut node, then the final representation for this node (less the child pointer), will be $(type, d, \delta, rt) = (1, 0, 2, 0)$.

5.2.6 Packet Classification

To classify a packet p, ByteCuts searches each $T_i \in \mathcal{T}$ in sequence. If the priority of the best match found so far exceeds that of T_i, then we do not need to search T_i as none of its rules can be of a higher priority than those already found. Since $\mathcal{T}$ is sorted based on priority, this means that we can stop searching.

To search a tree T, we first look at its root node and determine which type of node it is. If it is a leaf node or split node, we search these in exactly the same manner as HyperSplit [20]. Otherwise, it is a cut node. We right-shift $p[d]$ over and then extract the rightmost δ bits. We interpret this as an index into the child array, which we then search recursively.

Consider classifying packet (3, 0). For the first tree, we have a cut node with $d = 0$, $\delta = 2$ and $rt = 0$ as shown in Section 5.2.5. If we right shift 3 by 0 bits and then extract the last 2 bits, we get 3 (or 11_2). This points to a leaf containing r_4 and r_5. We see that r_4 does not match this packet but r_5 does. We do not need to search the second tree because r_5 has a higher priority than both r_8 and r_9. We thus conclude that r_5 is the best matching rule.

5.3 Theoretical Results

In this section, we prove some theoretical results about ByteCuts. For these proofs, we make four assumptions.

1. Each good tree (those containing only good rules and no bad rules) leaves at most $\chi \cdot n_L$ rules unselected.

2. The number of bad trees produced is constant.

3. At least some fraction β of the cuts on m rules contain no more than $\alpha \cdot m$ rules in the largest child node, for constants $0 < \alpha, \beta < 1$.

4. The replication factor is at most $\rho \geq 1$.

Theorem 1. *Given assumptions 1 and 2, the maximum number of trees produced is constant.*

Proof. From assumption 1, if we start with n rules, then after k trees are produced, at most $n_r = n \cdot \chi^k$ rules remain. Additionally, ByteCuts stops producing good trees when $n_r \leq \tau \cdot n$. Thus $\tau \cdot n \geq n \cdot \chi^k$. Solving for k we get that $k \log \chi \leq \log \tau$ and $k \leq \log(\tau)/\log(\chi)$. Since we can only produce an integer number of tables, $k \leq \lceil \log(\tau)/\log(\chi) \rceil$. Since τ and χ are constants, then so is k. Since both the number of good trees and the number of bad trees are constant, so is the total number of trees. □

If we set $\chi = 0.2$ and $\tau = 0.05$, then we expect 2 good trees. If we assume that there will be 1-3 bad trees (such as for non-TCP/UDP traffic and wide source or destination addresses), then we expect 3 to 5 trees total. In Section 5.4, we find experimentally that the precondition to Theorem 1 usually holds and that we achieve these expected results.

We use Theorem 1 to compute the expected search and construction times.

Corollary 2. *Given assumptions 1, 2, and 3, the search time for ByteCuts is* $O(\log n)$.

Proof. Similar to a quicksort proof, we can use assumption 3 to show that the maximum tree height will be at most $\beta \log_\alpha n$. Since Theorem 1 shows the number of trees is constant, the total search time is $O(\log n)$. □

Theorem 3. *Given all four assumptions, the construction time for ByteCuts is* $O(dw \cdot \rho n \log n)$.

Proof. There are two costs to building the classifier: selecting the rules and building the tree; tree construction is more expensive. Building a tree node requires considering each pair of d fields and w lengths. For each such pair, we must build a key for each rule. From assumption 4, there are at most ρn rules shared between all of the nodes of a given level. Each level of the tree thus costs $O(dw \cdot \rho n)$. From Corollary 2, we know that the height of a tree is $O(\log n)$ and from Theorem 1 we know that the number of trees is constant. Thus, the total construction time for ByteCuts is $O(dw \cdot \rho n \log n)$. □

5.4 Experimental Results

In this section, we compare ByteCuts against existing methods including HyperCuts [22], HyperSplit [20], and SmartSplit [7].

5.4.1 Experimental Setup

We ran our experiments on rule lists created with ClassBench [29]. ClassBench includes 12 different seed files in 3 different categories: 5 access control lists (ACL), 5 firewalls (FW) and 2 IP-chains (IPC). Seeds in the same category can have very different properties. In addition, we have 18 real-life rule lists ranging in size from 100 rules to 7600 rules that we used for timing comparisons.

From ClassBench, we generated rule lists of 7 different sizes. For each size, we generated 5 classifiers for each seed. This yields 60 classifiers per size and 420 classifiers total.

We compare ByteCuts to HyperCuts, HyperSplit, and SmartSplit using implementations provided by the authors. For each method, we try leaf sizes (*binth* in the original papers, equivalent to n_{leaf}) of 8 and 16 and report the better result. We leave the other variables

at their default values.

We measure six metrics: classification time, number of trees, tree height, memory accesses, construction time, and memory usage. Classification time, tree height, and memory accesses all are measures of classification speed. Classification time measures average classification speed as it represents how long an algorithm takes to process an actual packet trace. Tree height and memory access instead measure the worst case classification speed. Tree height estimates the worst cost of traversing each tree, but not the linear search of rules in the leaves. Memory accesses estimates the combined cost of both the trees and the rules in the leaves. For multiple tree methods, such as ByteCuts and SmartSplit, these sums are upper bounds; it may not be possible for an individual packet to be compared to the deepest rules in every tree.

For each metric, we report the mean value and standard deviation of a classifier across all of the rule lists under consideration (such as for a specific rule list size). We also compare algorithm A to ByteCuts by reporting the ratio A/BC (usually as a percentage).

We ran these experiments on a computer with an Intel Xeon CPU @ 2.8 Ghz, 4 cores, and 6 GB of RAM running Ubuntu 16.04 and compiled with gcc 5.4.0. We verified the correctness of ByteCuts and SmartSplit by ensuring each classifier returns matching results for each test packet.

5.4.2 Classification Time

We compare the mean packet classification time of ByteCuts and SmartSplit in Figure 5.9. We do not consider HyperCuts and HyperSplit because the provided implementations do not support classification (they report the static qualities of the created tree) and because they were unable to produce a tree given the time and memory constraints for many rule

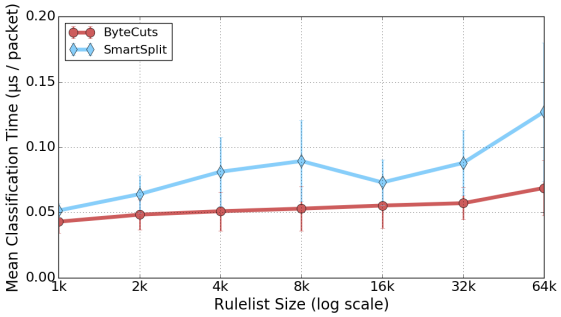

Figure 5.9: Classification Time

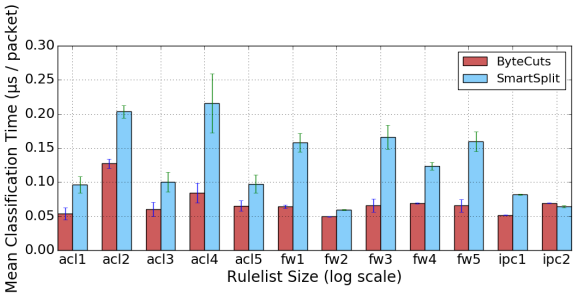

Figure 5.10: Classification Time on 64k rules

lists.

We measure classification time by generating a stream of 1 million packets, measuring how long each classifier takes to classify the whole stream, and then computing the average time per packet. Each method receives the same packet stream.

Overall, we find that ByteCuts performs 58% better than SmartSplit. The improvement increases as the rule list becomes larger; on 1k rules, ByteCuts and SmartSplit perform

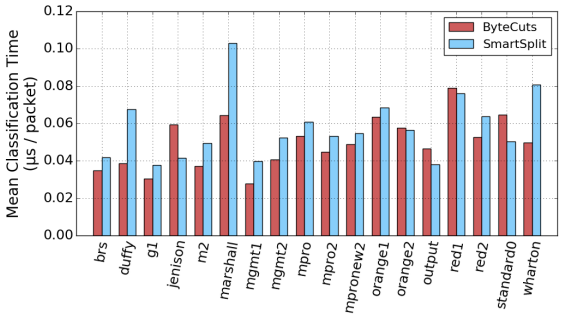

Figure 5.11: Classification Time on real-life rules

roughly equally well, but on 64k rules, ByteCuts performs 84% better. On the 64k rule lists, SmartSplit requires an average of $0.127\mu s$ per packet, but ByteCuts requires only $0.069\mu s$ per packet.

We also analyze the classification time across each of the individual seeds. For 11 of the 12 seeds, ByteCuts performs better than SmartSplit. SmartSplit is only faster than ByteCuts on the IPC2 seed, and it is only 6.9% faster. This is smaller than the difference for every other seed; the next smallest difference is that SmartSplit is 19.9% slower for the FW2 seed. In extreme cases, such as the ACL4 seed, SmartSplit is 155% slower than ByteCuts.

These results are consistent with the experiments on the real-life rule lists. On these rule lists, ByteCuts is faster than SmartSplit on 14 of the 18 rule lists. Overall, ByteCuts averages 0.049 μs / packet while SmartSplit requires 0.056 μs / packet. These values both fall in the range between the averages for our 1k experiments and our 8k experiments, which is roughly the sizes of our real-life classifiers.

Finally, ByteCuts has much less variance than SmartSplit. For example, the standard deviation of SmartSplit across the 64k rule lists is 2.5 times that of ByteCuts. This makes ByteCuts more consistent and predictable.

5.4.3 Number of Trees

We measured the number of trees required by both ByteCuts and SmartSplit. Since HyperCuts and HyperSplit always produce a single tree, they are not shown. The results can be seen in Figure 5.12.

We find that ByteCuts produces a relatively stable number of trees, averaging at 3.52 trees across all of the rule lists. Instead of varying by the number of rules, it is dependent on the qualities of the individual seed as shown in Figure 5.13. By contrast, SmartSplit uses

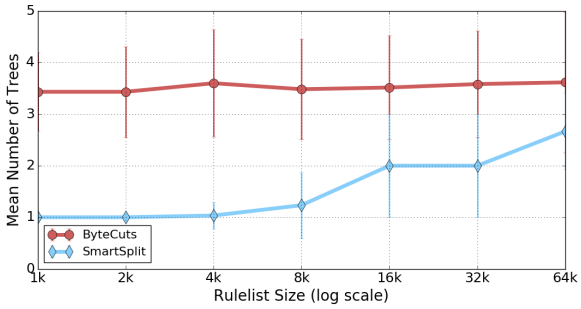

Figure 5.12: Number of trees

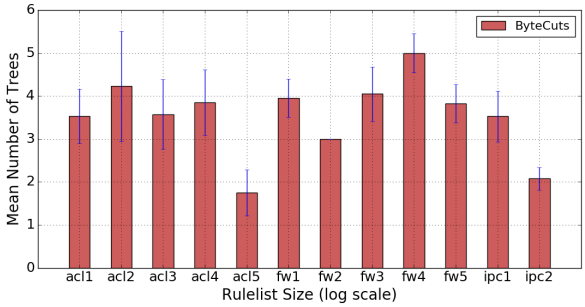

Figure 5.13: Number of trees across seeds

more tables when the number of rules increases.

These results match the theoretical results from Theorem 1 which predicts that the number of trees should be constant. This suggests that the assumptions made in Section 5.3 are valid for our classifiers.

5.4.4 Tree Height

We compare the mean tree height of all four methods in Figure 5.14. Since ByteCuts and SmartSplit produce multiple trees, we report the performance of each method with stacked bars. The lowest bar represents the average height of the first tree, the second bar the second tree, and so on. For ByteCuts this is an upper bound on the actual worst-case as it may not be possible to hit all of the deepest branches simultaneously.

The height of each individual ByteCuts tree is significantly smaller than for the other

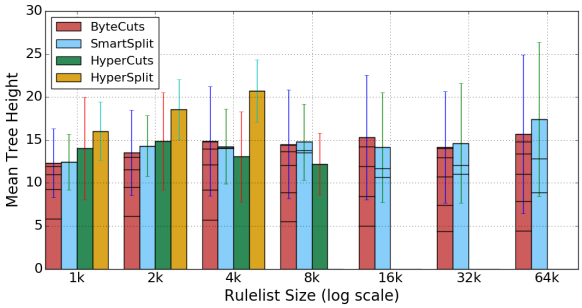

Figure 5.14: Mean Tree Height

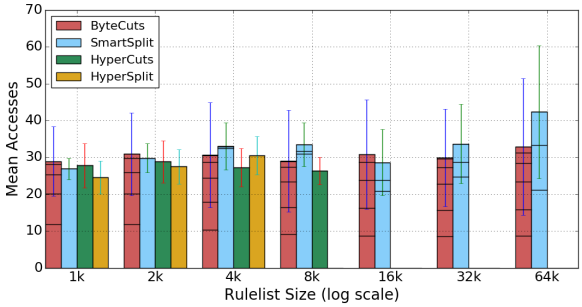

Figure 5.15: Mean Memory Accesses

methods, but the total height is similar to the other methods. On the 1k rule lists, the sum height of the ByteCuts trees is 1% smaller than those of SmartSplit. This improves to 10% for the 64k rule lists.

Comparing individual trees, we find that SmartSplit's first tree tends to be, on average, 2.1 times the size of ByteCuts's. This ratio decreases as the number of rules increases because SmartSplit produces more (generally smaller trees). For the 64k rule lists, where SmartSplit produces multiple trees, this reduces to only 1.59 times on average.

If a high-priority match is found, it is not necessary to search later trees (if they only contain lower-priority rules) as no rule can have a higher priority. This is thus a significant advantage for ByteCuts as it may only need to look at a single, smaller, tree. This explains why ByteCuts performs so much better than SmartSplit.

We also compare the number of accesses (tree height + leaf size) in Figure 5.15. As before,

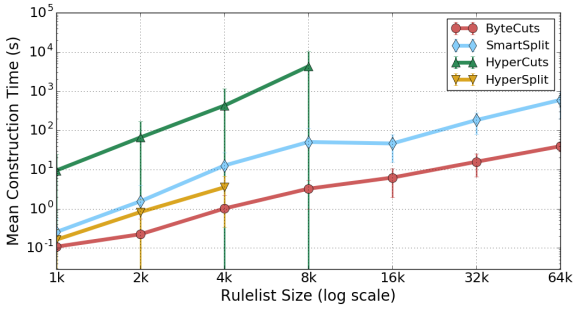

Figure 5.16: Mean Construction Time

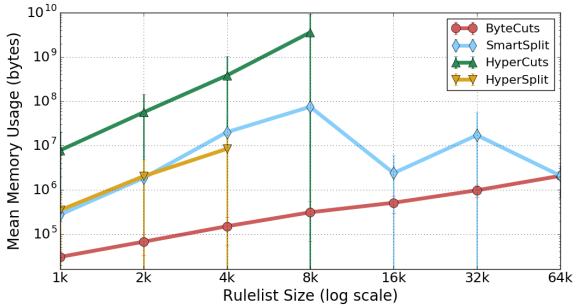

Figure 5.17: Mean Memory Usage

we represent the accesses per tree with stacked bars. Again, each individual ByteCuts tree is significantly smaller than the other individual trees and the sum is similar to the other methods. We find that when SmartSplit produces multiple trees, the resulting sum is larger than the corresponding sum for ByteCuts.

Unlike the classification time metric, which measures the expected average case time, the tree height and number of accesses measure the theoretical worst case time. Under an adversarial model where a classifier continuously receives the worst possible packets for it, our results show that ByteCuts is competitive with all the other algorithms in the worst case and outperforms SmartSplit as the number of rules increases.

5.4.5 Construction Time

ByteCuts has significantly faster construction times than any of the other methods, finishing in under a minute even on the largest rule lists. SmartSplit is the only one of the other algorithms that was able to finish on all of the rule lists. Both HyperCuts and HyperSplit have problems with time and memory for larger rule lists (16k for HyperCuts and 8k for HyperSplit). Of the two, HyperSplit has faster construction times; its times are faster than SmartSplit for smaller rule lists, but it is still slower than ByteCuts.

5.4.6 Memory Usage

We compare the mean memory usage for each classifier in Figure 5.17. This includes all of the memory for tree nodes, arrays, and the rule list.

ByteCuts requires multiple orders of magnitude less memory than the other methods, with HyperCuts having the worst requirements by an order of magnitude. ByteCuts's tree separation scheme allows it to effectively minimize rule replication, keeping its memory usage modest. HyperCuts and HyperSplit do not have any way to limit rule replication. Additionally, the wide branching factor of HyperCuts causes rule replication to run rampant.

HyperSplit has problems with certain incompatibilities. Some rules cannot be split on either address field, which is the primary separator for the other rules. These rules are thus replicated into nearly every leaf. If there are too many of these bad rules (in particular if it exceeds *binth*), they need to be separated individually. This can potentially take $O(n_{bad}^{d})$ cuts, yielding a sudden memory explosion. For this reason our data for HyperSplit only goes to 4k rules; some seeds exceeded the memory limits for each of the larger sizes.

SmartSplit has erratic memory requirements depending on whether it chose to build one

or multiple trees and if those were HyperCuts or HyperSplit trees. This is why its curve oscillates.

5.4.7 Summary of Trade-offs

In packet classification there is an inherit trade-off between time and memory. Multi-tree packet classifiers, such as ByteCuts and SmartSplit, balance this trade-off primarily by adjusting the number of trees produced. Different rules will have different properties. For example, one rule may use the source address field while another will use the destination address field instead. Placing both rules into the same tree will result in rule replication as any cut that can differentiate one rule will divide the other rule into multiple parts. This in turn increases both memory consumption and tree height.

Creating multiple trees allows rules with different properties to be separated. This reduces the amount of rule replication required which directly reduces memory consumption. This can be most easily seen by comparing Figure 5.12 and Figure 5.17; memory consumption normally increases with the number of rules, but when SmartSplit produces multiple trees instead of a single tree, its memory consumption decreases instead.

Producing more trees can both increase and decrease the search times required. With multiple trees, each individual tree tends to be smaller than before, so each individual tree is faster to search. However, collectively they may take more time to search than a single tree. Thus a small number of trees may be faster than a single tree, but additional trees may increase the time required. This both explains the dip in SmartSplit's classification time in Figure 5.9 and is one of the reasons that ByteCuts is faster than SmartSplit.

Chapter 6

PartitionSort

In Chapter 5, we looked at a software-based searching method for traditional offline packet classification. With modern software-defined networks (SDN), such as OpenFlow [17], packet classification has become an online problem. By this, we mean that rather than simply classifying packets, the classifier must also be capable of modifying itself to match a changing rule list. For many traditional packet classifiers, ByteCuts included, the only solution is to completely rebuild the classifier. For SDN, this solution is infeasible.

In this chapter, we will instead develop an online software-based packet classifier which we call PartitionSort. Like before, our approach will involve separating the rule list into multiple trees. Unlike before, our trees will not directly partition rules spatially. Instead, we will establish a partial order over the rule list. Our partition strategy will then ensure that each sublist is totally ordered. This allows each sublist to be binary searched.

6.1 Related Work

6.1.1 Independent Set

In [26], the authors propose a method for dividing the rule list into sets where all of the rules in a given set are disjoint in a particular field. Since the rules are non-overlapping in this field, the rules can be sorted based upon it. This allows the list to be binary searched

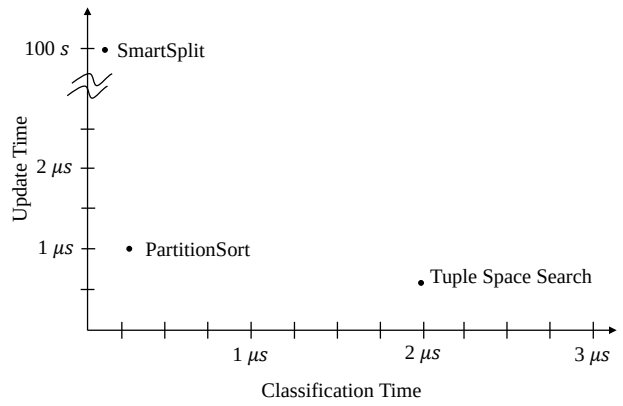

Figure 6.1: PartitionSort is uniquely competitive with prior art on both classification time and update time. Its classification time is almost as fast as SmartSplit and its update time is similar to Tuple Space Search.

and guarantees a search time for any given tree. Unfortunately, the number of sets is large enough that their sum cost is prohibitive. PartitionSort generalizes this method with a more effective comparison function and other enhancements that produces fewer sets.

6.2 PartitionSort Overview

PartitionSort combines the speed of traditional decision tree classifiers with the sortability strategy of Independent Set [26] to produce a software packet classifier that is competitive with both high-speed classification methods on classification time and fast update methods on update time as shown in Figure 6.1. We generalize the sortability criterion from [26] into multiple dimensions. PartitionSort partitions a rule list into a small number of sortable rule lists. We store each sortable rule list in a multi-key binary search tree leading to $O(d+\log n)$ classification and update time per tree. The memory requirement is linear in the number of rules.

This method requires us to overcome four technical challenges. The first is *defining sortability for multi-dimensional rules.* It is challenging to sort multi-dimensional rules in

a way that helps packet classification. We address this challeng by describing a necessary requirement that *any* useful ordered rule list must satisfy and then introduce our field order comparison that progressively compares multiple rules one field at a time.

The second technical challenge is *designing an efficient data structure for sortable rule lists* that supports high-speed classification and fast update using only linear space. We show that our notion of rule list sortability can be translated into a multi-key set data structure and hence it is possible to have $O(d + \log n)$ search / insert / delete time using a multi-key balanced binary search tree. We also propose a path compression optimization to speed up classification time.

The third technical challenge is *effectively partitioning a non-sortable rule list into as few sortable rule lists as possible*. We address this challenge by a reduction to a new graph coloring problem with geometric constraint. Then, we develop a fast offline sortable rule list partitioning algorithm that runs in milliseconds.

The forth challenge is *designing an effective online partitioning algorithm*. This is particularly challenging because we must not only achieve fast updates but also preserve fast classification time in the face of multiple updates. We use an offline rule list partitioning algorithm as a subroutine to design a fast online rule list partitioning algorithm that still achieves fast classification time.

6.3 Rule Sortability

We must overcome two main challenges in defining sortable rule lists. The first is that we must define an ordering function $\leq$ that allows us to order rules which are d-dimensional hypercubes. Consider the two boxes r_1 and r_4 in Figure 6.2; one might argue that $r_1 \leq r_4$

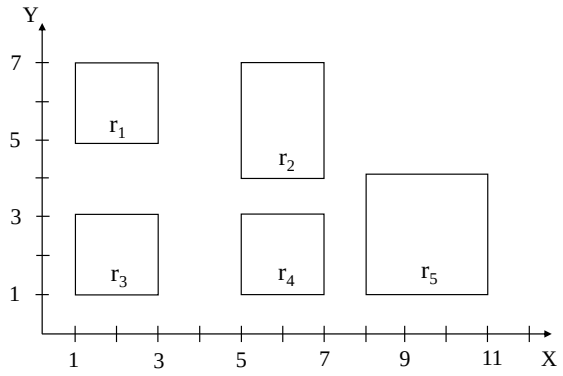

Figure 6.2: Example of ordered rule list with $\leq_{xy}$

since r_1's projection in the X dimension is smaller than r_4's projection in the X dimension; however, one could also argue that $r_4 \leq r_1$ because r_4's projection in the Y dimension is smaller than r_1's projection in the Y dimension. The second challenge is that $\leq$ must be useful for packet classification purposes. Specifically, suppose $\leq$ defines a total ordering on rules r_1 through r_n and we are trying to classify packet p. If we compare packet p to rule r_i and determine that $p > r_i$, then packet p can only match some rule among rules r_{i+1} through r_n. That is, packet p must not be able to match any of rules r_1 through r_i.

6.3.1 Field Order Comparison Function

We propose the following *field order comparison function* for sorting rules. The intuition comes from the fact that sorting one-dimensional rules is straightforward. The basic idea is that we choose a field order (permutation) $\vec{f}$ and then compare two rules by comparing their projections in the current field until we can resolve the comparison. Continuing the example from above, if we choose field order XY then $r_1 \leq_{XY} r_4$ because $[1, 3]$, r_1's projection in field X is smaller than $[5, 7]$, r_4's projection in field X. A more interesting example is that $r_4 \leq_{XY} r_2$ because their projections in field X are both $[5, 7]$ which forces us to resolve the ordering using their projections in field Y.

We now formally define our field order comparison function. Let $\vec{f}_i$ denote the ith field of $\vec{f}$. For any rule r, let $\vec{f}_i(r)$ denote the projection of r in $\vec{f}_i$, that is its component in field $\vec{f}_i$.

We now define how to compare two intervals using $\vec{f}$. For any two intervals $s_1 = [\min(s_1), \max(s_1)]$ and $s_2 = [\min(s_2), \max(s_2)]$, we say that $s_1 < s_2$ if $\max(s_1) < \min(s_2)$. We say that two intervals are not comparable, $s_1 \parallel s_2$ if they overlap but are not equal. Together this defines a partial ordering on intervals. To illustrate, $[1, 3] < [4, 5]$, but $[1, 3] \parallel [3, 5]$ as they overlap at point 3.

We define the field order comparison function $\leq_{\vec{f}}$ based on field order $\vec{f}$ as follows.

Definition 1 (Field Order Comparison)**.** *Consider two rules r_i and r_j and a field order $\vec{f}$. If $\vec{f}_1(r_i) \parallel \vec{f}_1(r_j)$, then $r_i \parallel_{\vec{f}} r_j$. If $\vec{f}_1(r_i) < \vec{f}_1(r_j)$, then $r_i <_{\vec{f}} r_j$. If $\vec{f}_1(r_i) > \vec{f}_1(r_j)$, then $r_i >_{\vec{f}} r_j$. Otherwise, $\vec{f}_1(r_i) = \vec{f}_1(r_j)$. If $\vec{f}$ contains only one field, then $r_i =_{\vec{f}} r_j$. Otherwise, the relationship between r_i and r_j is determined by $\vec{f}' = \vec{f} \setminus \vec{f}_1$. That is, we eliminate the first field and recursively compare r_i and r_j starting with field $\vec{f}_2$ which is $\vec{f}'_1$.*

To simplify notation, we often use the field order $\vec{f}$ to denote its associated field order comparison function $\leq_{\vec{f}}$ and we drop $\vec{f}$ from $\leq$ if it is clear from context.

For example, consider again the rectangles in Figure 6.2. We have $r_1 \leq r_2$ since $X(r_1) < X(r_2)$ and X is the first field. We also have $r_3 \leq r_1$ because $X(r_3) = X(r_1)$ and $Y(r_3) \leq Y(r_1)$. The set of 5 rectangles are totally ordered by XY; that is $r_3 < r_1 < r_4 < r_2 < r_5$. On the other hand, the set of 5 rectangles are not totally ordered by YX because $r_4 \parallel r_5$.

We now define sortable rule lists.

Definition 2 (Sortable Rule List)**.** *A rule list L is sortable if and only if there exists a field order $\vec{f}$ such that L is totally ordered by $\vec{f}$.*

6.3.2 Usefulness for Packet Classification

We now prove that our field order comparison function is useful for packet classification. We first must define how to compare a packet p to a rule r. Using the same comparison function does not work because we want p to match r if the point defined by p is contained within the hypercube defined by r, but $\leq_{\vec{f}}$ would say $p \parallel r$ in this scenario. We thus define the following modified comparison function for comparing a packet p with a rule r using field order $\vec{f}$.

Definition 3 (Packet Field Order Comparison)**.** *If $\vec{f_1}(p) < \min(\vec{f_1}(r))$, then $p <_{\vec{f}} r$. If $\vec{f_1}(p) > \max(\vec{f_1}(r))$, then $p >_{\vec{f}} r$. If $\vec{f}$ contains only one field, then p matches r. Otherwise, the relationship between p and r is determined by $\vec{f'} = \vec{f} \setminus \vec{f_1}$.*

Lemma 4. *Let n d-dimensional rules r_1 through r_n be totally ordered by field order $\vec{f}$ and let p be a d-dimensional packet. For any $2 \leq i \leq n$ if $p < r_i$ then $p > r_j$ for $1 \leq j < i$. Likewise, for any $1 \leq i \leq n-1$, if $p < r_i$ then $p < r_j$ for $i < j \leq n$.*

Proof Sketch. We discuss only the first case where $p > r_i$ as the other case is identical by symmetry. Consider any packet p and rules r_i and r_j where $p > r_i$ and $r_i > r_j$. Applying Definitions 1 and 3, we can prove that transitivity holds and $p > r_j$. □

6.4 Multi-dimensional Interval Tree (MITree)

We describe an efficient data structure called a Multi-dimensional Interval Tree (MITree) for representing a sortable rule list given field order $\vec{f}$. In particular, we show that MITrees achieve $O(d + \log n)$ time for search, insertion, and deletion; we refer to this as $O(d + \log n)$ *dynamic time*.

	X	Y	Z
r_1	[0,2]	[0,6]	[1,2]
r_2	[3,4]	[0,1]	[1,2]
r_3	[3,4]	[2,3]	[3,5]
r_4	[3,4]	[4,5]	[3,6]
r_5	[5,6]	[0,5]	[1,2]
r_6	[5,6]	[0,5]	[4,6]
r_7	[5,6]	[0,5]	[7,9]

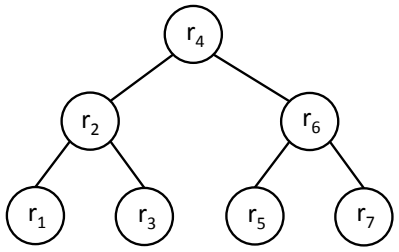

Figure 6.3: A sortable rule list with 7 rules and 3 fields, field order $\vec{f} = XYZ$, and the corresponding simple binary search tree where each node contains d fields.

An obvious (and naive) approach is to store an entire rule for each node and construct a balanced binary search tree because the rules are totally ordered by field order $\vec{f}$. The running time is clearly $O(d \log n)$ because the height of a balanced tree is $O(\log n)$ and each comparison requires $O(d)$ time. The problem is this running time is not scalable to higher dimensions. Figure 6.3 shows an example sortable rule list with its corresponding naive binary search tree.

We can speed up the naive approach by a factor of d by using one field at a time with a clever balancing heuristic. We show this in two steps. First we show $O(d + \log n)$ dynamic time for a special case where all intervals are points. We extend the solution to the general case of sortable rule lists.

The special case where all intervals are points is exactly a multi-key dictionary problem. A multi-key dictionary is a dynamic set of objects that have comparisons based on a lexical order of keys. One example of a multi-key dictionary is a database table of n rows and d columns. There exists a multi-key binary search tree supporting $O(d + \log n)$ dynamic time for the multi-key dictionary problem [3, 30].

In the general case of sortable rule lists, we show that the $O(d+\log n)$ result is attainable. First, all keys are compared in lexical order because, by definition of sortable rule lists, field

order $\vec{f}$ is essentially a permutation of the natural order $[1..d]$. Furthermore, it is immediate by the definition of sortable rule lists that if two rules are identical in $[f(1), f(2), \ldots, f(i-1)]$ then at $f(i)$ they either overlap fully or have no intersection. This property allows us to construct an MITree that behaves as if the interval is only a point. Hence, we can use the same balancing heuristic to achieve $O(d + \log n)$ dynamic time. The search procedure is identical to point keys except that we need additional checking for each interval tree node.

These results yield the following Theorem:

Theorem 5. *Given a sortable rule list, a Multi-dimensional Interval Tree (MITree) supports searches, insertion, and deletion in $O(d + \log n)$ time using only linear space.*

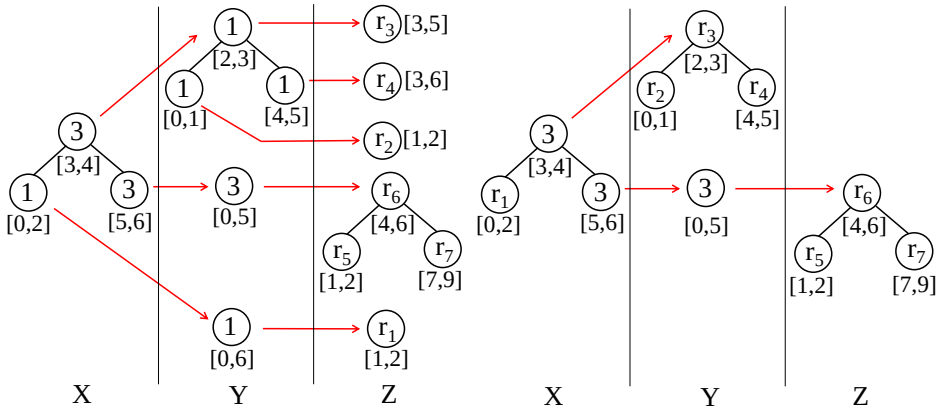

Figure 6.4: We depict the uncompressed (left) and compressed (right) MITree that correspond to the sortable rule list from Figure 6.3. Regular left and right child pointers are black and do not cross field boundaries. Next field pointers are red and cross field boundaries. In the compressed MITree, nodes with weight 1 are labeled by the one matching rule.

We perform one additional path compression optimization. We remove duplicate intervals for each field in the MITree so that each unique interval is shared by multiple rules. Each node v thus represent a particular interval in a given field. Each v thus has a third child containing the rules that match on v. Figure 6.4 shows an MITree for the rule list in Figure 6.3. Let $c(v)$ be the number of rules that match node v. We observe that c decreases from

earlier fields to later fields. Thus if $c(v) = 1$ then there is only one rule r that can match. In this case, we store the remaining field of r with v as shown in Figure 6.4. For the rest of this chapter, we use MITree to refer to a compressed MITree.

6.5 Offline Sortable Rule List Partitioning

Rule lists are not necessarily sortable, so we must partition rule lists into a set of sortable rule lists. Because the classification time depends on the number of sortable rule lists after partitioning, our goal is to develop efficient sortable rule list partitioning algorithms that minimize the number of sortable rule lists.

We first describe efficient and effective offline algorithms based on a reduction to a graph coloring problem called Field Order Graph Coloring. By offline, we mean that all the rules to be partitioned are know *a priori*. We start by observing that the offline rule list partitioning problem is essentially a coloring of an intersection graph (to be defined shortly) by the $d!$ possible field orders for sortable rule lists.

Formally, the basic approach is to partition rule list L into χ ordered rule lists $(L_i, \vec{f}(i))$ for $1 \leq i \leq \chi$ where $\bigcup_{i=1}^{\chi} L_i = L$, $L_i \cap L_j = \emptyset$ and each sublist L_i is sortable under field order $\vec{f_i}$. There is no relation between field orders $\vec{f}(i)$ and $\vec{f}(j)$; they may be the same or different. Let L consist of n, possibly overlapping, d-dimensional rules v_1 through v_n. For each of the $d!$ field orders, $\vec{f}(i)$ for $1 \leq i \leq d!$, we define the corresponding interval graph $G_i = (V, E_i)$ where $V = L$ and $(v_j, v_k) \in E_i$ if $v_j \parallel_{\vec{f}(i)} v_k$ for $1 \leq j, k \leq n$. We then define the interval multi-graph $G = (V, \bigcup_{i=1}^{d!} E_i)$ where each edge is labeled with the field order that it corresponds to. Our sortable rule list partitioning problem is that of finding a minimum coloring of V where the nodes with the same color form and independent set in

G_i for some $1 \leq i \leq d!$; the minimum number of colors is denoted $\chi(G)$.

Our offline partitioning framework is to find a maximal independent set of *uncolored* nodes in some G_i. We "color" these nodes which means we remove them from the current rule list to construct an MITree. We then repeat until all nodes are colored. We now describe how to find a large number of rules that are sortable in one of the $d!$ possible field orders.

We use the weighted maximum interval tree from [8] to compute a maximum independent set for any G_i in $O(dn \log n)$ time. Since there are $d!$ possible field orders, finding the maximum set out of all of the field orders is infeasible. Instead, we propose a greedy approximation, GrInd, that finds a maximal independent set and rules in time polynomial in both n and d. GrInd achieves this by avoiding full recursion for each subproblem and choosing $\vec{f}$ and the maximal independent set simultaneously.

We now describe GrInd which uses d rounds given d fields. In the first round, we have one set of rules which is the complete set of rules. GrInd selects the first field in the field order as follows. For each field f, it projects all rules onto the corresponding interval in field f. It then weights each unique interval by the number of rules that project onto it plus 1. We "add one" to bias GrInd towards selecting more unique intervals. GrInd then computes a maximum weighted independent set (using the method in [8]) of these unique weighted intervals. The first field is the field f that results in the largest weighted independent set. GrInd then eliminates all other intervals and their associated rules.

This process is repeated until all fields have been used or every set has only one rule. The remaining rules are sortable under field order $\vec{f}$ and ready to be represented by an MITree. Pseudocode for GrInd is shown in Algorithm 1. The running time is $O(d^2 n \log n)$ as shown in Theorem 6 below.

We give an example of GrInd in action in Figure 6.5. The input set of boxes with two fields

Algorithm 1: GrInd$(R, \vec{f})$

Input: Rule list R
Output: A maximal sortable rule list R' and field order $\vec{f'}$

```
L = [R]
f' = []
for i = 1 to d do
    for unselected field f do
        foreach rule list ℓ in L do
            let S be the set of projections from ℓ onto f
            let w_s = 1+ num rules with projection s ∈ S
            let S_ℓ = MaxWeightIndSet(S, w)
        let weight w_f = Σ_{ℓ∈L} |S_ℓ|
    add f with max weight to f'
    remove from each ℓ ∈ L any r where f(r) ∉ S_ℓ
    subpartition each ℓ ∈ L based on projection f(r)
let R' equal one rule from every list in L
return (R', f')
```

is shown in Figure 6.5 (a). For both possibilities of first fields, we create the corresponding intervals and the corresponding rules. In this example, the maximum weight choice is to use field X as we can get four rules with a weight of 6 (4 rules plus 2 partitions). If we use field Y instead, the maximum weight choice is only 3 rules of total weight 5. We then proceed with the remaining rules r_1, r_2, r_6 and r_7 as illustrated in Figure 6.5 (b). We process both partitions independently and see that we can choose all 4 rules to get a final result of 4 rules.

Theorem 6. *GrInd computes a field order $\vec{f}(i)$ for $1 \leq i \leq d!$ and a maximal independent set L_i for G_i such that L_i is sortable on $\vec{f}(i)$. GrInd requires $O(d^2 n \log n)$ time.*

Proof. The correctness follows from choosing a maximum weighted independent set in each round. For round $1 \leq i \leq d$, for each of the $d-i+1$ choices for fields in round i, the running time of computing the maximum weighted independent set is $O(n \log n)$. In later rounds, we likely have fewer than n boxes or rules left, but in the worst case for running time, all n

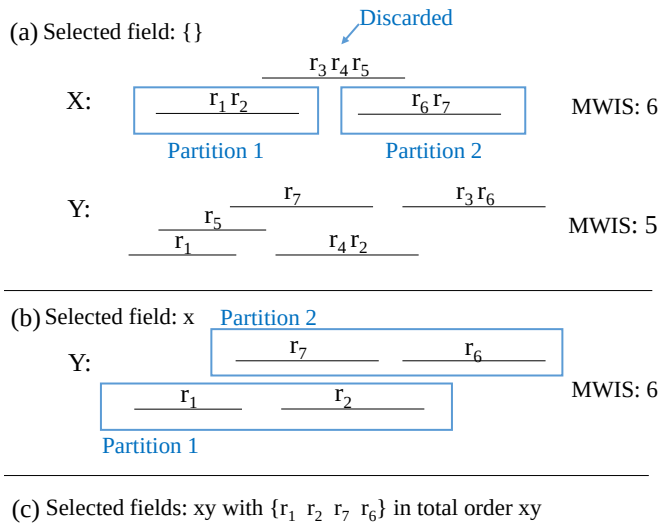

Figure 6.5: Example execution of GrInd.

boxes are considered in each round. This leads to the $O(d^2 n \log n)$ running time. □

6.6 PartitionSort: Putting it all together

In this section, we complete the picture of our proposed packet classifier, PartitionSort. We present a fully online rule partitioning scheme where we assume the rule list is initially empty and rules are inserted and deleted one at a time. We first describe how PartitionSort manages multiple MITrees. We then describe our rule update mechanism which still achieves a small number of partitions. We conclude by describing how PartitionSort classifies a packet given multiple MITrees.

To facilitate fast rule update and packet classification, PartitionSort sorts the multiple MITrees by their maximum rule priorities. Let $\mathcal{T}$ be the set of trees, $t = |\mathcal{T}|$, $pr(T)$ be the maximum priority in tree $T \in \mathcal{T}$, and $\vec{f}(T)$ be the field order of $T \in \mathcal{T}$. PartitionSort sorts $\mathcal{T}$ so that for $1 \leq i \leq j \leq t$, $pr(T_i) > pr(T_j)$. To maintain this sorted order, PartitionSort uses the following data structures. First a lookup table $M : L \to \mathcal{T}$ identifying which tree each rule is in. Second, for each tree $T \in \mathcal{T}$, PartitionSort maintains a heap $H(T)$ of the

priorities in T.

6.6.1 Rule Update (Online Sortable Partitioning)

Given an existing set of rules, we now describe how PartitionSort performs rule updates (insertions and deletions). We start with the simpler operation of deleting a rule r. Using M, we identify which table T contains r. We then delete r from T, remove $pr(r)$ from $H(T)$ and resort $\mathcal{T}$ if necessary. We use insertion sort since $\mathcal{T}$ is otherwise sorted. If no rules remain in T we delete T from M and from $\mathcal{T}$.

We now consider the more complex operation of inserting a rule r. We face two key challenges: choosing a tree $T_i \in \mathcal{T}$ to receive r and choosing a field order for T_i. We consider trees $T_i \in \mathcal{T}$ in priority order starting with the highest priority tree. When we consider T_i, we first see if T_i can receive r without modifying $\vec{f}(T_i)$. If so, we insert r into T_i. However, before committing to this updated tree, if $|T_i| \leq \tau$ for some small threshold such as $\tau \leq 10$, we run GrInd (cf. Section 6.5) on rules in T_i (now including r) and use a new field order if GrInd provides one partition with a different field order. This reconstruction of T_i will not take long because GrInd is fast and τ is small. If so, we update the field order and use the new tree T_i'. If we cannot insert r into T_i, we reject T_i as incompatible and move on to the next $T_i \in \mathcal{T}$. If no existing tree T_i will work, we create a new tree T_i for r with an empty heap $H(T_i)$. In all cases, we clean up by inserting $pr(r)$ into $H(T_i)$ and resort $\mathcal{T}$ if necessary.

6.6.2 Packet Classification

We now describe packet classification in the presence of multiple MITrees. We keep track of hp, the priority of the highest priority matching rule seen so far. Initially, $hp = 0$ to denote

no matching rule has been found. We sequentially search T_i for $1 \leq i \leq t$ for packet p. Before searching T_i, we first compare $pr(T_i)$ to hp. If $pr(T_i) < hp$, we stop the search process and return the current rule as no unsearched rule has a higher priority than hp. Otherwise, we search T_i for p and update hp if a higher priority matching rule is found. If we reach the end of the list, we either return the highest priority rule or we report that no rule was found. This priority technique was used in Open vSwitch [19] to speed up their implementation of TSS.

We argue that priority optimization benefits PartitionSort more than other methods. The reason for this is that our partitioning scheme is based on finding maximum independent sets iteratively. This produces partitions where almost all of the rules are in the first few partitions. In our experiments, we measured the percentage of rules contained in the first five partitions for both PartitionSort and TSS. This data is highlighted in Table 6.1 below. To summarize, we find that 99% or more of the rules in all classifiers reside in the first five partitions when ordered by maximum priority. In contrast, TSS exhibits a more random behavior where only some classifiers have many rules in the first five partitions.

32k	Average #Partitions			% rules in first five partitions/tuples		
	$\|G\|$	$\|O\|$	$\|T\|$	%G	%O	%T
acl	16.4	18.0	241.4	99.68	99.52	44.71
fw	22.3	25.0	99.2	99.20	99.12	10.18
ipc	9.5	10.1	170.9	99.92	99.88	25.99

Table 6.1: GrInd (G), Online (O) and TSS (T) average number of partitions (tuples) and percentage of rules in five partitions with highest priority rules

6.7 Experimental Results

6.7.1 Experimental Methods

We compare PartitionSort to two classification methods: TupleSpace Search (TSS) and SmartSplit (SS). TSS is the de facto standard packet classifier in OpenFlow classification because it has the fastest updates. It is a core packet classifier in Open vSwitch. SmartSplit is the fastest decision tree method as it analyzes the rule list to then choose between HyperSplit and HyperCuts. Like EffiCuts, it may construct multiple decision trees. We also compare PartitionSort to SAX-PAC to assess the effectiveness of our partitioning strategies. Our implementation of PartitionSort is available at GitHub[1].

We use the ClassBench utility [29] to generate rule lists. These rule lists are designed to mimic real rule lists. It includes 12 parameter files that are divided into three different categories: 5 access control lists (ACL), 5 firewalls (FW) and 2 IP chains (IPC). We generate lists of 1K, 2K, 4K, 8K, 16K, 32K, and 64K rules. For each size, we generate 5 classifiers for each parameter file, yielding 60 classifiers per size and 420 classifiers total.

We also consider the effect of the number of fields on the packet classifier. We modified ClassBench to output rules with multiple sets of 5 fields. Although this does not necessarily model real rule lists with more fields, it does allow us to project how our algorithms would perform given rules with more fields. We generate rule lists with 5, 10, 15, and 20 fields and 64K rules. As before, we generate 5 classifiers for each parameter file yielding a total of 20 rule lists per parameter file or 240 rule lists in total.

Our TSS implementation is the one from GitHub described in Pfaff *et al.* [19]. Specifically, we use their hash table implementation in the TSS scheme. We use their Priority TSS

[1] https://github.com/sorrachai/PartitionSort

implementation that searches tuples in decreasing order of maximum priority.

For SmartSplit, we use an implementation that is written by the authors of the paper which is hosted on GitHub[2].

We implemented SAX-PAC based on their paper [9] as there is no implementation available. We used the Coin-LP library [21] to select the order-independent sets; this is the current best algorithm for maximizing the size of each selected order-independent set for general d. We can use the fractional solution to lower-bound the number of sets required by SAX-PAC.

For all comparisons, we run PartitionSort with a fully online partitioning scheme unless explicitly stated otherwise; that is, the classifiers are generated by starting with an empty classifier and then repeatedly inserting rules. We perform internal comparisons between online and offline partitioning strategies.

We use four standard metrics: classification time, update time, space, and construction time. Space is simply the memory required by the classifier. We measure classification time as the time required to classify 1,000,000 packets generated by ClassBench when it constructed the corresponding classifier. To focus on classification time, we omit caching and consider only slow path packet classification on the first packet from each flow. We measure update time as the time required to perform one rule insertion or deletion. We perform 1,000,000 updates for each classifier as follows. We begin with each classifier having half the rules. We then perform a sequence of 500,000 insertions intermixed with 500,000 deletions choosing the rule to insert or delete uniformly at random from the currently eligible rules. Finally, we report all data by averaging over our rule lists. When we compare with TSS, for each unique rule list size and each number of fields and every metric, we average

[2]https://github.com/xnhp0320/SmartSplit

together all matching rule lists of the same type: ACL, FW, and IPC. When we compare with SmartSplit, for each unique rule lsit size and each number of field and every metric, we average together all matching rule lists. We then report[3] the average value of the metric, and, in some cases, a box plot of the metric.

All experiments are run on a machine with Intel i7-4790k CPU@4.00 GHz, 4 cores, 32 KB L1, 256KB L2, 8 MB L3 cache respectively, and 32 GB of DRAM. Most of the experiments were run on Windows 7. The exception is the SmartSplit Test; both SmartSplit and PartitionSort were run on Ubuntu 14.04.

To ensure correctness, we generate a stream of 1 million packets per rule list and test that all of the classifiers agree on how to classify them. All of the classifiers agreed for every input, so it is extremely probably that all of them work correctly.

6.7.2 PartitionSort versus TSS

We first compare PartitionSort with TSS. We compare these algorithms using the metrics of classification time, update time, construction time, and memory usage. We note that both algorithms have extremely fast construction times taking less than a second to construct all rule lists.

6.7.2.1 ClassificationTime

Our experimental results show that, on average, PartitionSort classifies packets 7.2 times faster than TSS for all types and sizes of rule lists. That is, for each of the 420 rule lists, we computed the ratio of TSS's classification time divided by PartitionSort's classification

[3]To minimize size-effects from hardware and the operating system, we run 10 trials for each of the classification/update time and report the average

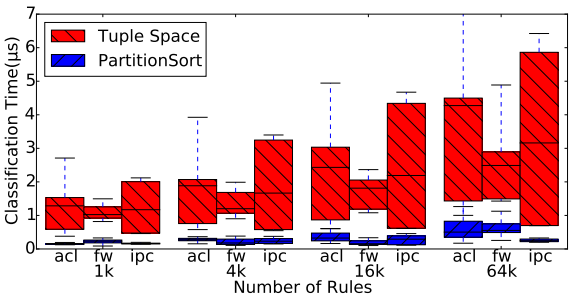

Figure 6.6: Classification Time vs Tuple Space Search

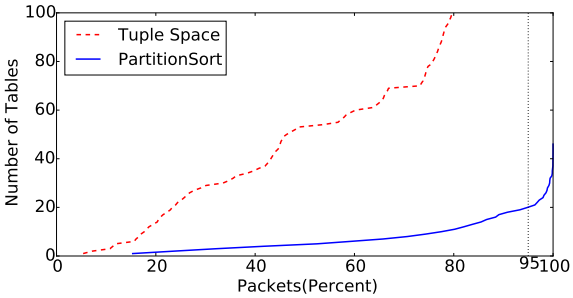

Figure 6.7: Partitions/Tuples Queried. A CDF distribution of number of partitions required to classify packets with priority optimization (cf. Section 6.6.2).

time and then averaged these 420 ratios. When restricted to the 60 size 64k rule lists, PartitionSort classifies packets 6.7 times faster than TSS with a maximum ratio of 20.1 and a minimum ratio of 2.6. Figure 6.6 shows the average classification times for both PartitionSort and TSS across all rule list types and sizes. If we invert these classification times to get throughputs, PartitionSort achieves an average throughput of 4.5 Mpps (million

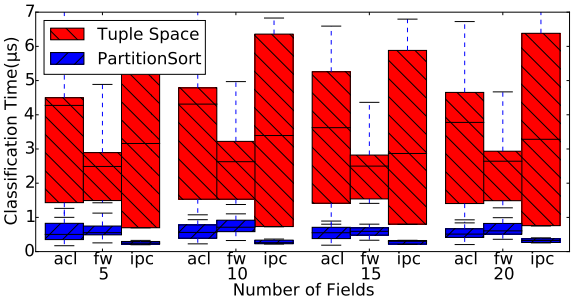

Figure 6.8: Classification Time on 64K Rules

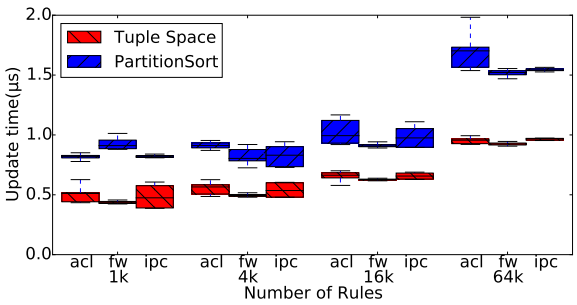

Figure 6.9: Update Time vs Tuple Space Search

packets per second) whereas TSS achieves an average throughput of 0.79 Mpps. We note 4.5 divided by 0.79 is less than 7.2; this is because taking an average of ratios is different than a ratio of averages.

Besides being much faster than TSS, PartitionSort is also much more consistent than TSS. Specifically, the quartiles and extremes are much tighter for PartitionSort than for TSS where the higher extremes do not even fit on the scale in Figure 6.6.

The reason for these factors is that PartitionSort requires querying significantly fewer tables than TSS. As seen in Figure 6.7, PartitionSort needs to query fewer than 20 tables 95% of the time, while TSS needs to only 23% of the time. Since search times should be roughly proportional to the number of tables queried, search times should be similar to the area under their respective curves. Since TSS's area is significantly larger, so too should their search times.

Both methods are relatively invariant in the number of fields, as seen in Figure 6.8. Because there are more field permutations available, PartitionSort is able to pick the best one for slightly bigger partitions. This results in a slight decrease in lookup times.

6.7.2.2 Update Time

Our experimental results show that PartitionSort achieves very fast update times, with an average update time of $0.65\mu s$ *and a maximum update time of* $2.1\mu s$. This should be fast enough for most applications. For example, PartitionSort can handle several hundred thousand updates per second. Figure 6.9 shows the average update times across all rule list types and sizes for both PartitionSort and TSS.

As expected, TSS is faster at updating than PartitionSort, but the difference is not large. Similar to classification time, we compute the ratio of PartitionSort's update time divided by TSS's update time for each rule list and then compute the average of those ratios. On average, PartitionSort's update time is only 1.7 times larger than TSS's update time. Restricted to the 64k rule lists, this average ratio is also 1.7. The data from Figure 6.9 does show that PartitionSort has an $O(\log n)$ update time.

6.7.2.3 Construction Time

Our experimental results show that PartitionSort has very fast construction times, with an average construction time of 83 ms for the largest classifiers. This is small enough that even significant changes to the rule list cause only minor disruption to the packet flow. As expected, TSS builds faster than PartitionSort, but the difference is not large. On average, PartitionSort's construction time is only 1.9 times larger than TSS's construction time. Restricted to the 64k rule lists, this average ratio decreases to only 1.4.

6.7.2.4 Memory Usage

Our experimental results show that PartitionSort requires less space than TSS. Both are space-efficient, requiring $O(n)$ memory, with PartitionSort requiring slightly less space than

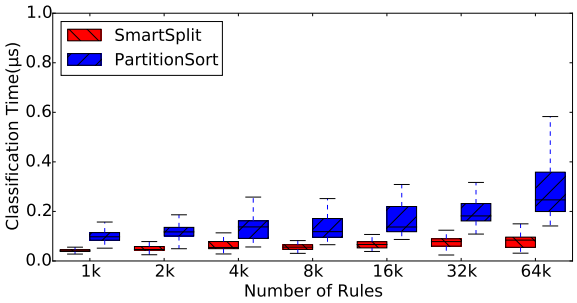

Figure 6.10: Classification Time vs SmartSplit

TSS.

6.7.3 Comparison with SmartSplit

We now compare PartitionSort with SmartSplit. We compare these algorithms using the metrics of classification time, construction time, and memory usage. We do not compare update time because SmartSplit does not support updates.

6.7.3.1 Classification Time

Our experimental results show that, on average, PartitionSort takes 2.7 times longer than SmartSplit to classify packets. This can be seen in Figure 6.10. There are two factors leading to this result. First, SmartSplit uses fewer trees than PartitionSort (the same advantage that PartitionSort has over TSS). Second, the wider branching factor provided by the HyperCuts trees used by SmartSplit reduces the overall tree height. While both classifiers have logarithmic search times, these two factors make SmartSplit's constant factor smaller.

6.7.3.2 Construction Time

PartitionSort can be build faster than SmartSplit by several orders of magnitude. This becomes increasingly more pronounced as the number of rules increases, where PartitionSort

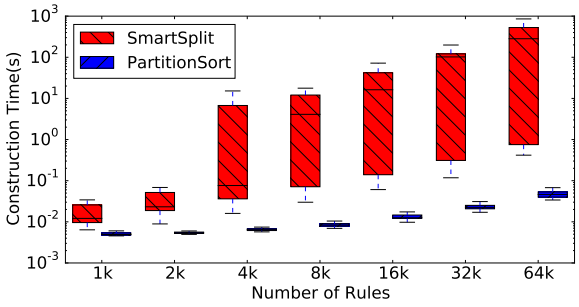

Figure 6.11: Construction Time in logarithmic scale.

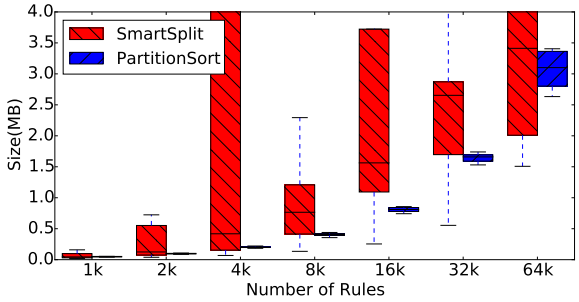

Figure 6.12: Memory Usage vs SmartSplit

takes less than a second but SmartSplit requires almost 10 minutes. This can be seen in Figure 6.11. Additionally, SmartSplit does not support updating. Together, this means that SmartSplit is not appropriate for cases where the rule list is updated frequently.

6.7.3.3 Memory Usage

Our experimental results show that PartitionSort requires less space than SmartSplit. The average space requirements for each rule list size is shown in Figure 6.12. SmartSplit's memory usage is very erratic, depending on whether it decided to make one tree or several, and whether those trees are HyperCuts or HyperSplit trees. SmartSplit tends to select single HyperCuts trees when the rule lists are smaller and multiple HyperSplit trees when larger. This causes the apparent drop in memory usage as the number of rules increases. We expect that memory usage will continue to rise dramatically as rule replication again becomes a

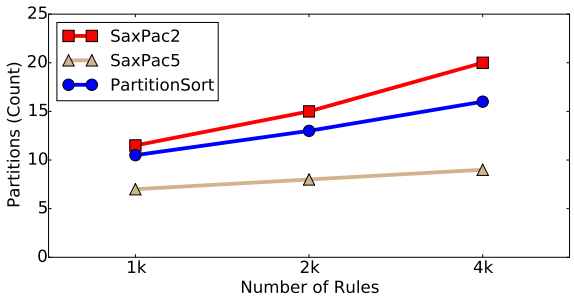

Figure 6.13: Number of Partitions vs SAX-PAC

problem, but no further remedies remain for preventing it.

6.7.4 Comparison with SAX-PAC

The number of partitions required by PartitionSort is competative with SAX-PAC. When using the same number of fields, PartitionSort requires 1.7 times more tables than SAX-PAC. However, the classifier used by SAX-PAC requires $O(\log^{d-1} n)$ time. For this reason, they recommend using only 2 fields for order-independent partitioning. This significantly increases the number of partitions required. In this case, SAX-PAC requires 1.4 times more tables than PartitionSort. These results cna be seen in Figure 6.13.

SAX-PAC is too expensive to construct. Most of the algorithms used by SAX-PAC require $O(dn^2)$ time [9]. Additionally, their set-cover approximation requires $O(n^2)$ space. Thus, even 4000 rules can take hours to build, making it infeasible to use. For this reason, Figure 6.13 only goes up to 4000 rules.

6.7.5 Comparison of Partition Algorithms

We now compare our offline (cf. Section 6.5 and online (cf. Section 6.6) partitioning algorithms. Figure 6.14 shows the number of partitions required by each version.

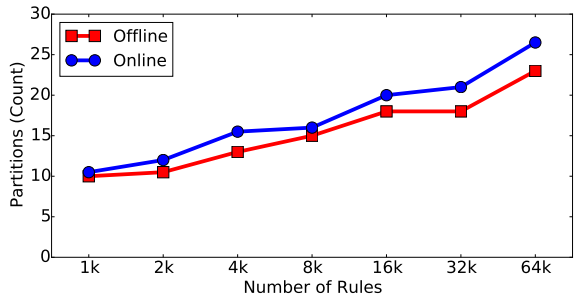

Figure 6.14: Number of Partitions for Offline and Online PartitionSort

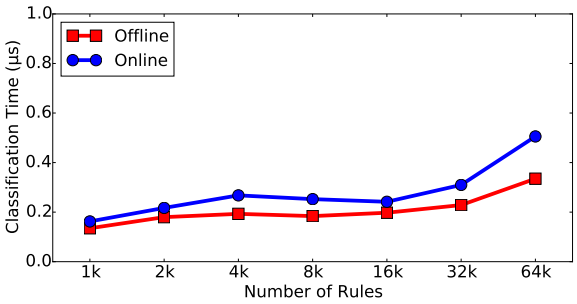

Figure 6.15: Classification Time for Offline and Online PartitionSort

Online partitioning performs almost as well as offline partitioning. Online partitititioning requires only 17% more trees than offline partitioning. This translates into 31% larger classification times as seen in Figure 6.15. We conclude that a single offline construction time at the beginning with online updates should be more than sufficient for most purposes.

Chapter 7

TupleMerge

In Chapter 6, we developed an online tree-based packet classifier. This classifier is both reasonably fast at classifying packets while also supporting frequent updates. However, PartitionSort is hampered by two problems. First, the $O(d + \log n)$ search times of the trees limits the maximum classification speed. Second, Tuple Space Search already has market deployment.

In this chapter, we present a hashing-based online packet classifier, which we call TupleMerge. These hash tables have $O(d)$ search times, allowing it to be faster than PartitionSort, both for classification and updates. Additionally, the classifier produced is compatible with that of Tuple Space Search which can ease deployment.

Rule	Source	Dest	Tuple
r_1	000	111	(3, 3)
r_2	010	100	(3, 3)
r_3	011	100	(3, 3)
r_4	010	11*	(3, 2)
r_5	10*	101	(2, 3)
r_6	110	***	(3, 0)
r_7	***	011	(0, 3)

Table 7.1: Example 2D rulelist

Rule	Source	Dest	Tuple
r_1	000	111	(3, 3)
r_2	010	100	(3, 3)
r_3	011	100	(3, 3)
r_4	010	11^*	(3, 2)
r_5	10^*	101	(2, 3)
r_6	110	$***$	(3, 0)
r_7	$***$	011	(0, 3)

Table 7.2: TSS builds 5 tables

7.1 Related Work

7.1.1 Tuple Space Search

Tuple Space Search (TSS) is one of the oldest offline packet classification methods that was left behind as faster methods were developed [24]; TSS has reemerged for online packet classification because it supports fast updates. In particular, Open vSwitch uses TSS as its packet classifier because TSS offers constant-time updates and linear memory [17].

In TSS, each partition is defined by a corresponding mask tuple $\vec{m}$ which represents the actual bits used in each of the prefix or ternary fields of all rules in that partition. Since all the rules in a partition have the same $\vec{m}$, we can store these rules using a hash table where the masked bits $\vec{m}$ for each rule form the hash key. During classification, those same bits are extracted from the packet for the search key. For example, consider $r_5 = (10*, 101)$ from Table 7.1. We produce the hash key $h_5 = hash(10, 101)$ and store $h_5 \rightarrow r_5$. Later, suppose we wish to query for packet $(5, 5) = (101, 101)$ Extracting the same bits, we get the search key $(10, 101)$. We then search for the key $hash(10, 101)$ and get r_5 as a match.

We now describe how to manage the multiple tables that result from partitioning. To speed up the search process, we sort the tables by the highest priority rule each table contains

and search them in this order. This allows us to avoid searching a table if we have found a matching rule that exceeds the maximum priority of any rule in that table. This speeds up the searches for the case that a high priority matching rule is found in an early table, but it does not help for the cases where the correct rule has low priority or no matching rule is found.

For example, consider the rules in Table 7.1 would be placed in 5 tables, which can be seen in Table 7.2. Rules r_1, r_2, and r_3 each use all of the bits and are placed together. The other rules all have unique tuples and so each gets its own table. If we search for packet $(110, 011)$ then T_4 returns r_6 as a match. Since this has a higher priority than r_7, it is not necessary to search T_5 even though r_7 is also a match.

The main drawback of TSS is that the number of partitions or tables is large. This results in relatively slow packet classification as many tables must be searched. We improve upon this by putting rules from multiple TSS tables into a single hash table. This results in many fewer hash tables.

Srinivasan *et al.* suggest using a (usually 1-dimensional) preclassifier to identity which tables contain rules that might match the packet [24]. This allows fewer tables to be searched, potentially resulting in significant time savings. The tradeoff is that the preclassifier requires extra time and memory; for some rule lists, it may take more time to query the preclassifier than is saved by reducing the number of tables queried. In contrast, TupleMerge reduces the total number of tables and does not need a preclassifier.

Another option for reducing the number of tables is to reduce the number of tuples by ignoring certain fields (such as the port fields). This results in a drastic reduction in the number of tables which is exponential in the number of fields. However, some rules may be identical on the reduced fields; extra effort must be spent to separate these rules. We

also exploit this optimization which, combined with other optimizations, results in far fewer tables than TSS.

Yet another strategy to avoid querying hash tables is to use Bloom filters [23]. Bloom filters are a very space-efficient probabilistic hash set that can answer whether an item is probably in the set (with the probability being related to memory usage) or definitely not in the set. Since Bloom filters are space efficient, they can be stored in cache and used to check whether a matching rule probably exists before performing a query to slower memory. This is a complementary optimization that TupleMerge can also leverage; in fact, it may be more efficient for TM as TM has fewer hash tables than TSS.

7.2 Online Algorithms

We now describe our TupleMerge (TM) classifier that significantly improves upon TSS [24] by significantly reducing the number of hash tables required to represent the classifier. Since the search time is proportional to the number of tables searched, this results in a significant improvement in search times. Additionally, since our method does not use any sort of preclassifier, we are not penalized by losses elsewhere.

We first describe the two key differences that allow TM to outperform TSS. We then review the classification process. Finally, we describe how to update the classifier during rule changes.

7.2.1 Key Differences

There are two key differences that make TupleMerge faster at packet classification than TSS. First, TM exploits the observation that not all bits are always required to identify which rules

are potential matches. In TSS, rules are separated so that rules that use exactly the same tuple will end up in the same hash table and every other rule will end up in a different table. Many rules have similar but not identical tuples which TSS places in separate tables. TM allows these rules to be placed into the same table, reducing the number of tables required; this leads to faster classification.

Second, TM has methods for reducing the number of hash collisions. It is possible for two keys to be hashed to the same value. Additionally, both TSS and TM may produce the same hash key from different rules because of information that is omitted. For TSS, this is because most implementations will omit port fields (which are ranges) and sometimes other fields as well; the exact field selection is usually chosen by the designer ahead of time and is the same for all tables. For TM this occurs because it has decided to omit some bits from the tuple. TSS has only one option when this happens; it must sequentially search all of the colliding rules until it finds a match. TM uses the number of collisions to control the number and specificity of its tables; if there are too many collisions (determined by parameter c) it adds more specific tables to reduce the number of collisions so that no table will have more than c rules collide.

We now explain how TM allows rules with different tuples to be placed in the same hash table. Like TSS, each TM hash table T has an associate tuple $\vec{m}_T$. In TM, unlike TSS, a rule r may be placed in T if $\vec{m}_T \subseteq \vec{m}_r$; in TSS, r may be placed in T only if $\vec{m}_T = \vec{m}_r$. We create a hash key $k_{T,r}$ from $\vec{m}_T$ and r by extracting the bits matching $\vec{m}_T$ from r, similarly to TSS. The key issue is identifying what tuples $\vec{m}_T$ should be constructed for each ruleset, particularly as we construct the classifier in an online fashion. We describe this in more detail in our Rule Insertion section below.

For example, in Table 7.1 r_2 has tuple $(3, 3)$ and r_4 has tuple $(3, 2)$. We could place both

r_1 and r_4 into the same table if the table used $(3, 2)$ for its tuple. If we wanted to use $(3, 3)$ for the tuple, we would find that r_4 is incompatible since $3 > 2$; this would require splitting r_4 into two rules which we do not do because this complicates updating. Another possibility is to use $(3, 1)$ for the tuple; both rules would have a hash key of $(010, 1**)$ which creates a hash value collision in the table. This tuple would be allowed if the number of collisions in this hash value does not exceed c.

TM maintains a list $\mathcal{T}$ of tables created along with their tuples where the tables are sorted in order of the maximum rule contained and a mapping $F : L \to \mathcal{T}$ of rules to tables. During packet classification, only $\mathcal{T}$ is used. The mapping F supports fast deletion of rules. Both require space that is linear in n.

Standard online construction (where rules are received on at a time) is represented by a series of rule insertions as described below. If the rules are mostly known ahead of time, we use the offline construction scheme described in Section 7.3 to create a better classifier.

7.2.2 Packet Classification

TupleMerge classifies packets just like TSS classifies packets. As with TSS, once a high priority matching rule is found, later tables with lower priority rules do not need to be searched. While the table construction algorithm changes, the packet classifier can remain the same. This is a potential implementation advantage that allows TM to be easily integrated into existing implementations such as Open vSwitch that use TSS.

7.2.3 Tuple Selection and Rule Insertion

We now describe how TupleMerge chooses which tuples to use for tables. For TSS, no choice was necessary; we simply used the tuples defined by the used bits for each rule. For TM, we have the opportunity to use fewer bits to define tuples which can reduce the number of tables but may come at the cost of increasing the number of collisions. We strive to balance between using too few bits and having too many collisions and using too many bits and having too many tables. One key observation is that if two rules overlap, having them collide in the same table is a better option than creating separate tables for the two rules. For this reason, we maintain a relatively high collision limit parameter c.

Consider the situation when we have to create a new table T for a given rule r. This occurs for the first rule inserted and for later rules if it is incompatible with all existing tables. In this event, we need to determine $\vec{m}_T$ for a new table. Setting $\vec{m}_T = \vec{m}_r$ is not a good strategy; if another similar, but slightly less specific, rule appears we will be unable to add it to T and will thus have to create another new table. We thus consider two factors: is the rule more strongly aligned with source or destination addresses (usually the two most important fields) and how much slack needs to be given to allow for other rules. If the source and destination addresses are close together (within 4 bits for our experiments), we use both of them. Otherwise, we drop the smaller (less specific) address and its associated port field from consideration; r is predominantly aligned with on of the two fields and should be grouped with other similar rules. This is similar to TSS dropping port fields, but since it is based on observable rule characteristics it is more likely to keep important fields and discard less useful ones.

We then look at the absolute lengths of the addresses. If the address is long, we are more

likely to try to add shorter lengths and likewise the reverse. We thus remove a few bits from both address fields with more bits removed from longer addresses. For 32 bit addresses, we remove 4 bits, 3 for more than 24, 2 for more than 16, and so on (so 8 and fewer bits don't have any removed). We only do this for prefix fields like addresses; both range fields (like ports) and exact match fields (like protocol) should remain as they are.

Now consider the case where we need to insert a new rule r into TM's classifier. We first try to find an existing table that is compatible with r. We search the table $T \in \mathcal{T}$ in order of max priority rule. For each table T, r is compatible with T if $\vec{m}_T \subseteq \vec{m}_r$. If no compatible table is found, we create a new table for r as described above.

If a compatible table is found, we perform a hash probe to check whether c rule with the same key already exist within T. If not, we add r to T and add the mapping $r \rightarrow T$ to F. Otherwise, this is a signal that we ignored too many bits in $\vec{m}_T$ and we need to split the table into two tables. We select all of the colliding rules ℓ and find their maximum common tuple $\vec{m}_\ell$. Normally $\vec{m}_\ell$ is specific enough to hash ℓ with few or no collisions. We then create a new table T_2 with tuple $\vec{m}_\ell$ and transer all compatible rules from T to T_2. If $\vec{m}_\ell$ is not specific enough, we find the field with the biggest difference between the minimum and maximum tuple lengths for all of the rules in ℓ and set that field to be the average of those two values. We then transfer all compatible rules as before. This guarantees that some rules from ℓ will move and that T_2 will have a smaller number of collisions than T did.

7.2.4 Rule Deletion

Deleting a rule r from TupleMerge is straightforward. We can find which table T contains r with a single lookup from F. We then remove r from T and the mapping $r \rightarrow T$ from F. This requires $O(1)$ hash updates.

7.3 Offline Algorithms

Up until now, we have discussed online algorithms; here we discuss offline construction. When most of the rules are known *a priori*, we can use this extra information to make better decisions about how to partition the rules. This is relevant for several online scenerios including the case where we periodically rebuild the classifier assuming the rebuild process can be done in roughly a second. As noted earlier, OpenFlow hardware switches can require roughly half a second for the data plane to catch up to the control plane.

7.3.1 Table Selection

The main principle behind our table selection strategy is that if a high-priority rule is matched in an early table, then any future tables that contain only lower priority rules do not need to be searched. However, if a lower-priority match is found in an early table, we must still search later tables as they could contain a better match. To this effect, we try to place as many high-priority rule early to maximize the chances that we will not need to search later tables.

Let L_i be the first i rules in L. Let $\vec{m}_{L_i}$ be the maximum common tuple of these rules; that is the tuple that contains all of the bits that they have in common. We now create a counter H of hashes, ℓ_i, a proposed table, and $\bar{\ell}_i$, the rules not in ℓ_i. Fore each rule $r \in L$, we compute $h(r)$. If $H(h(r)) < c$, were c is an argument representing a maximum number of collisions, we add r to ℓ_i and increment $H(h(r))$. Otherwise we add r to $\bar{\ell}_i$. Each different $\vec{m}_{L_i}$ will yield a ℓ_i and $\bar{\ell}_i$. To maximize the probability that we do not need to search future tables, we select the ℓ_i that maximizes the maximum priority of $\bar{\ell}_i$ (the first missing rule is as late as possible), breaking ties by maximum $|\ell_i|$, thus reducing the number of tables required.

Rule	Source	Dest	Nominal Tuple	Used Tuple
r_1	000	111	(3, 3)	(3, 2)
r_2	010	100	(3, 3)	(3, 2)
r_3	011	100	(3, 3)	(3, 2)
r_4	010	11*	(3, 2)	(3, 2)
r_5	10*	101	(2, 3)	(2, 0)
r_6	110	***	(3, 0)	(2, 0)
r_7	***	011	(0, 3)	(0, 3)

Table 7.3: TupleMerge builds 3 tables

Rule	Source	Dest	Nominal Tuple	Used Tuple
r_1	000	111	(3, 3)	(3, 0)
r_3	011	100	(3, 3)	(3, 0)
r_4	010	11*	(3, 2)	(3, 0)
r_6	110	***	(3, 0)	(3, 0)
r_2	010	100	(3, 3)	(0, 3)
r_5	10*	101	(2, 3)	(0, 3)
r_7	***	011	(0, 3)	(0, 3)

Table 7.4: An alternate TupleMerge scheme builds 2 tables

We make a new table $T = (\ell_i, \vec{m}_{L_i})$ and append it to $\mathcal{T}$, and repeat on the remaining rules $L' = \bar{\ell}_i$ until L' is empty.

For example, consider the rules in Table 7.1. Tuple $(3, 2)$ can contain 4 rules without any collisions: r_1, r_2, r_3 and r_4. Tuple $(3, 3)$ can only contain the first three rules, while $(2, 2)$ has a collision between r_2 and r_3, making them worse choices. We create a table containing rules r_1, r_2, r_3, and r_4 and repeat on the remaining three rules. The results from this process can be seen in Table 7.3.

Other strategies exist that may yield fewer tables. For example, Table 7.4 contains only two tables. However, unless the incoming packet matches rule r_1, both tables will need to be searched. Table 7.3 will need to search only 1 table if any of r_1, r_2, r_3, or r_4 match which may give it better results in practice even if the worst case is not as good.

7.3.2 Table Consolidation

This strategy sometimes produces multiple tables that have the same tuple. When this happens, we merge the two tables together into a single table. This removes a table query while examining at most the same number of rules as before.

7.4 Experimental Results

In this section, we perform experiments to compare the effectiveness of TupleMerge against PartitionSort (PS), TupleSpace Search (TSS), SmartSplit (SS), and SAX-PAC (SP).

7.4.1 Experimental Setup

We ran our experiments on rule lists created with ClassBench [29]. ClassBench comes with 12 seed files that can be used to generate rule lists with different properties. While the seeds are divided into three categories (5 access control lists (ACL), 5 firewalls (FW) and 2 IP-chain (IPC)), different seeds in the same category can have very different properties.

We generate rule lists of nine different sizes, from 1k up to 256k rules. For each of the 12 ClassBench sees, we generate 5 rule lists of each size for 540 rule lists total.

We use these rule lists in three types of experiments. For an offline experiment, we construct a static data structure from all the rules and then use the data structure to classify packets. For an online experiment, we have two protocols. In the first protocol, we create a dynamic data structure where the rules are fed to the online algorithm one at a time in a random order with each algorithm receiving the rules in the same order. The resulting classifier is then used to classify packets. It classifies 1,000,000 packets which are generated using the ClassBench method for generating packets given a rule list. In the second protocol,

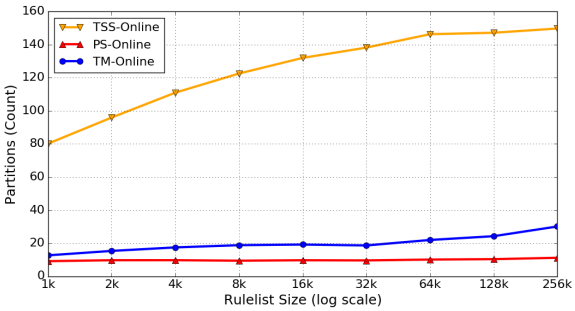

Figure 7.1: Average # of partitions required for each rulelist size

which we use to measure update time, half of the rules are selected at random to be in the initial active rule list. We create an initial data structure by inserting these rules one at a time into the data structure in a random order where each algorithm receives the rules in the same order. A list of 500,000 insertions and 500,000 deletions is then created and shuffled. Each insertion inserts one of the excluded rules and each deletion removes one of the included rules, each chosen at random from the available rules. This sequence is the same for each algorithm. In this protocol, we do not use the resulting data structure to classify packets.

For a given rule list L and a given algorithm A, we measure four things. First is classification time, denoted $CT(A, L)$, which is the total time required to classify all 1,000,000 packets divided by 1,000,000. Second is the number of partitions $P(A, L)$ in the resulting data structure. This is useful as a static estimator of each data structure's classification speed. Third is update time, denoted $UT(A, L)$, which is the total time required to perform all 1,000,000 rule updates divided by 1,000,000. Note we do not include the time to create the initial data structure with half the rules. Last is memory, denoted $M(A, L)$, which is the total memory required by the data structure.

For each metric, we average the results across all rule lists of a given size or seed. To better compare an algorithm A, typically PS, against TM, we compute the relative classification time of A and TM on a rule list L, denoted $RCT(A, TM, L)$, as the ratio

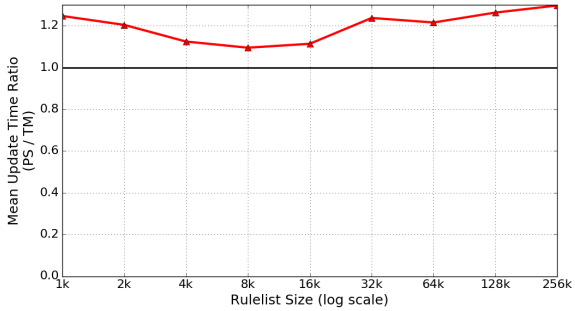

Figure 7.2: Relative Update Time between PS and TupleMerge

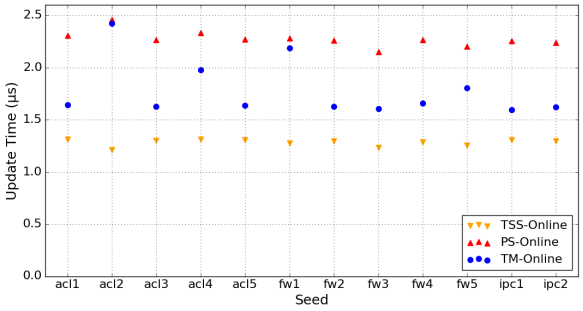

Figure 7.3: Absolute Update Time on 256k rules

$CT(A, L)/CT(TM, L)$; higher values are better for TM. Likewise, we compute the relative update time of A and TM for rule list L, denoted $RUT(A, TM, L)$ to be the ratio $UT(A, L)/UT(TM, L)$. We average these relative metrics across all rule lists of a given size or seed We note that for the two relative time metrics, the range of ratios is relatively small whereas for the raw metrics, some values are much larger than others and thus have a much larger impact on the final average. This leads to some apparent discrepancies between the two averages.

These experiments were run on a machine with an Intel Xeon CPU @ 2.8 GHz, 4 cores, and 6 GB of RAM running Ubuntu 16.04. We verified the correctness of classification results by ensuring each classifier returns the same answer for each test packet.

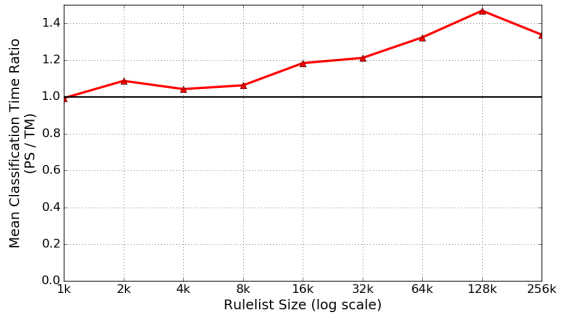

Figure 7.4: Relative Online Classification Time between PS and TupleMerge

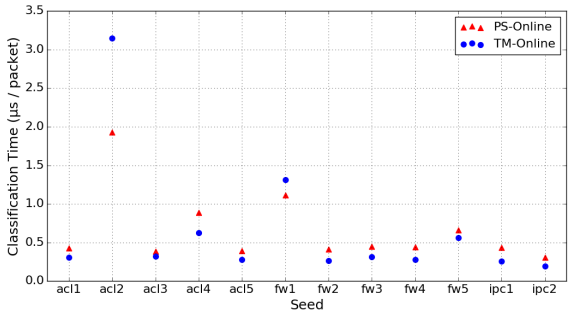

Figure 7.5: Absolute Online Classification Time on 256k rules

7.4.2 Comparison with PartitionSort

We compare TupleMerge against PartitionSort from Chapter 6 since PS is the state-of-the-art online packet classifier. We first compare their online versions and then compare their offline versions.

7.4.2.1 Online Comparison

In our experiments, online TM outperforms online PS in both classification speed and update time. Focusing on the relative metrics averaged over all seeds and sizes, **TM performs rule updates 30% faster than PS and TM classifies packets 34.2% faster than PS**. PS also requires 2.71 times the amount of memory that TM does, though both required only linear memory in the number of rules.

We now exam these rules in more detail. The number of tables required for online TM,

online PS and TSS are shown in Figure 7.1. Relative update time results average over each rule list size are shown in Figure 7.2 and raw update time results for online TM, online PS, and TSS for each seed for the 256k rule lists are shown in Figure 7.3. Relative classification time results average over each rule list size are shown in Figure 7.4 and raw classification time results for online TM and online PS for each seed for the 256k rule lists are shown in Figure 7.5.

Looking more closely at update time, we see that TM consistently outperforms PS in every way. Specifically, TM has a faster update time for all rule list sizes and for all 12 seeds. For the largest rule lists, TM outperforms PS for all 12 seeds and has an average update time of only $1.78\mu s$ whereas PS has an average update time of $2.27\mu s$.

Looking more closely at classification time, TM again outperforms PS for all rule list sizes with the gap between TM and PS generally increasing with rule list size, but we do observe that PS does outperform TM for some seeds. On the largest rule lists, TM takes $0.64\mu s$ on average to classify a packet, whereas PS takes $0.65\mu s$. TM classifies faster on 10 of the 12 seeds, but the two remaining seeds (ACL2 and FW1) are the slowest for both PS and TM which skews the average classification times to be closer than the average relative classification time. Focusing on the other 10 seeds, TM takes $0.15\mu s$ to $0.76\mu s$ (average $0.34\mu s$) to classify a packet and PS takes $0.26\mu s$ to $1.00\mu s$ (average $0.47\mu s$) to classify a packet. These results can be seen in Figure 7.5.

These results can be explained by the number of partitions required and the time required to search each partition. We plot the average number of partitions required by TM and PS for each rule list size in Figure 7.1. As expected, TM typically requires a few more partitions that PS. Focusing on the 256k rule lists, TM requires an average of 30 partitions whereas PS requires an average of 11 partitions. TM typically achieves a smaller classification time

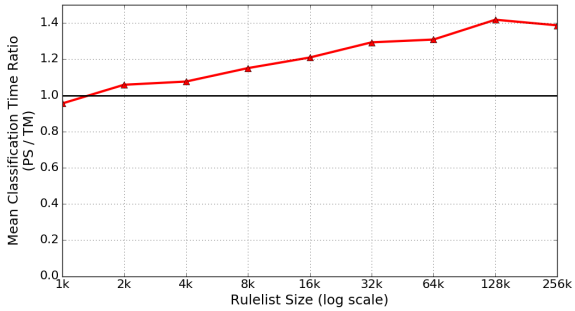

Figure 7.6: Relative Offline Classification Time between PS and TM

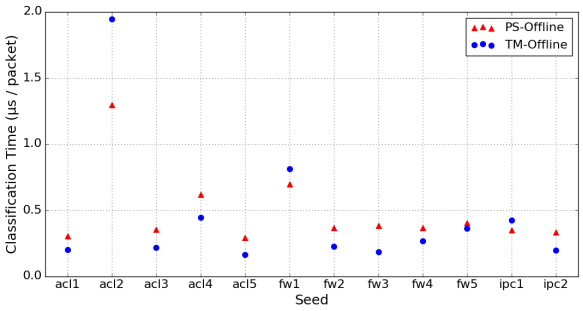

Figure 7.7: Absolute Offline Classification Time on 256k rules

because TM uses hash tables ($O(d)$ time) whereas PS uses trees ($O(d+\log n)$) time. However, for the worst case seeds, ACL2 and FW1, TM requires 131 and 53 partitions, respectively, whereas PS requires only 24 and 17 partitions, respectively. For the remaining ten seeds, TM requires at most 38 partitions.

The number of partitions has more effect on classification time than rule update time. For rule insertion, once we find an available partition, we do not need to search later partitions. For rule deletion, both schemes store a pointer to the partition containing each rule.

For memory, for the 256k rule lists, TM requires only 4.47 MiB on average whereas PS requires 12.19 MiB on average.

7.4.2.2 Offline Comparison

We now compare the offline versions of TM and PS focusing on classification time, memory usage, and construction time. *Offline TM again outperforms offline PS for all rule list sizes except the 1k rule lists, and the gap between TM and PS generally increases with rule list size.* However, we again observe that PS does outperform TM for some seeds. Focusing on the 256k rule lists, on average, offline TM classifies packets 39% faster than PS. These results can be seen in Figure 7.6. The outliers again are the two seeds ACL2 and FW1 where offline TM requires 78 and 47 partitions, respectively, whereas offline PS requires only 20 and 13 partitions, respectively. When comparing memory, the results are similar as those for the online versions. This is to be expected since the memory usage is tied to the number of rules rather than the number of tables. Finally, on average, offline TM builds its dynamic data structure 3.8 times faster than offline PS does. For the 256k rule lists, TM requires an average of $2.9s$ whereas PS requires an average of $9.0s$.

One of the main reason to consider offline TM and PS is to periodically rebuild the dynamic data structures to support faster packet classification and rule update. Averaging performance over all rule list sizes and all seeds, offline TM is 38% faster than online TM at classifying packets. Given that it takes an average of $2.9s$ to rebuild the dynamic data structures for a 256k rule list, this may be acceptable, particularly if the rebuild process can happen in parallel offline. We further not that existing hardware switches may be behind the controller by $400ms$ [10], so $2.9s$ may be acceptable.

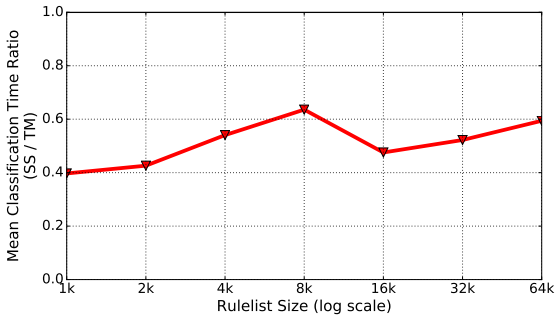

Figure 7.8: Relative classification time between SS and TupleMerge

7.4.3 Comparison with Tuple Space Search

We only compare online TM against TSS since TSS is inherently an online algorithm. *TM is, on average, 7.43 times faster at classifying packets than TSS, and TM is, on average, 39% slower at performing rule updates than TSS.* For our 256k rule lists, TM takes an average of $0.64\mu s$ to classify a packet whereas TSS takes an average of $2.93\mu s$ to classify a packet. For our 256 rule lists, TM takes on average $1.78\mu s$ to update a rule wheras TSS takes on average 1.28μ to update a rule. We thus see that for only a small penalty in rule update, TM achieves a significant gain in classification time. This improvement in classification time is largely explained by the fact that TM requires many fewer tables than TSS, as can be seen in Figure 7.1. Besides improving classification time, TM also requires 1.93 times less memory on average than TSS. Given the significant improvement in classification time and even memory, TM clearly is a better choice than TSS as an online packet classification algorithm.

7.4.4 Comparison with SmartSplit

We now compare offline TupleMerge with SmartSplit (SS), the current state-of-the-art offline packet classification algorithm, to assess how much we give up in classification time to achieve

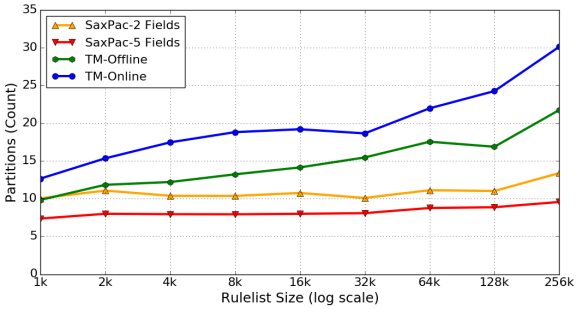

Figure 7.9: Partitions produced

fast updates. We compare these algorithms on classification time, construction time, and memory usage. The largest rule lists we use are the 64k rule lists because of SS's slow construction time.

SS is significantly faster than TM (or the other methods studied. For our 64k rule lists, on average, SS classifies packets in $0.12\mu s$ whereas offline TM classifies packets in $0.24\mu s$. The relative classification time required by SS and TM can be seen in Figure 7.8.

TM requires orders of magnitude less time to build a classifier than SS. Whereas TM can usually generate a classifier in seconds even for very large rule lists, SS can take hours, even for reasonably sized rule lists.

Finally, *SS has very unpredictable memory requirements.* For smaller rule lists, it produces a single HyperCuts tree, maximizing classification speed. As the number of rules increases, it switches over to multiple trees and/or HyperSplit trees, both of which reduce the rule replication and thus memory required. This doesn't fix the underlying problem though. Eventually, there are enough problematic rules that this is not enough.

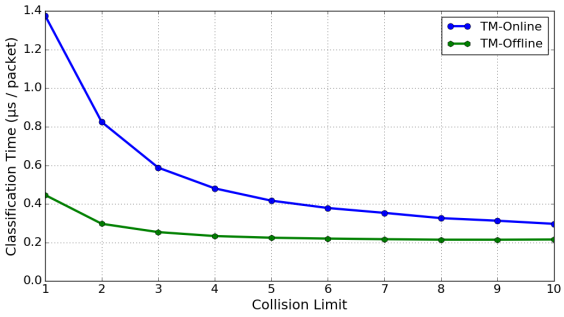

Figure 7.10: Smaller collision limit c

7.4.5 Comparison with SAX-PAC

SAX-PAC (SP) [9] splits the rule list into order-independent sets and then builds a classifier for each set. Since the order-independent sets are the most general type of collisionless rule list, this serves to lower-bound the number of partitions required for a variety of methods. We compare against them to estimate how well our table selection scheme performs.

TM requires a comparable number of partitions to SP. When SP uses all of the fields, TM requires on average 2.1 times more partitions than SAX-PAC. This shows that the number of partitions required by TM is not too much larger than the thoretical minimum. However, the classifier recommended by SP requires $O(\log^{d-1} n)$ time. To counteract this, they recommend using only 2 fields, which reduces the search cost to only $O(\log n)$. This increases the number of partitions that they require; TM requires only 68% more tables. These results can be seen in Figure 7.9. In addition, since they recommend using classifiers with a $O(\log^{d-1} n)$ time whereas TM's hash tables require $O(d)$ time, the advantage of fewer partitions disappears.

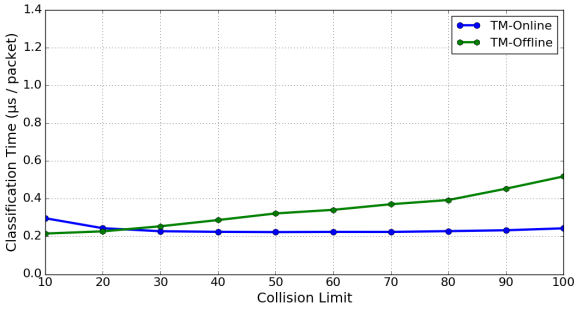

Figure 7.11: Larger collision limit c

7.4.6 TupleMerge Collision Limit

We now measure how the collision limit c influences TM's effectiveness. We show how the classification speed for both online and offline TM vary as a function of c in Figures 7.10 and 7.11. A higher c allows us to produce fewer tables, but a particular table may take longer to search since we must do a linear search of the colliding rules. *for online TM, classification speed is maximized with $c = 40$.* Smaller rule lists generally need smaller c and the incremental gain tapers off as c exceeds 20. Smaller c, such as $c = 10$, give acceptable results. The best classification speed can be seen in Figure 7.11. *The collision limit for offline TM is maximized earlier at $c = 8$.* This best speed can be seen in Figure 7.10.

Chapter 8

Conclusion

We presented four methods for packet classification. Each of these methods works by using smart division to divide the problem into simpler subproblems.

In Chapter 4, we presented Diplomat, a method for using dimensional reduction to compress rule lists into smaller equivalent lists. We presented both a general framework for Diplomat as well as specific resolvers and schedulers for both range and prefix fields. In a side-by-side comparison with ACL Compressor, Diplomat performed 30.6% better on large range-ACLs, 12.1% better on large prefix-ACLs, and 9.4% better on large mixed-ACLs.

In Chapter 5, we presented ByteCuts, a packet classifier that utilizes rule list separation to create multiple decision trees. It builds the trees using a new type of cut node that selects several consecutive bits from a packet and interprets it as an index into the child array. This allows it to classify packets 58% better than SmartSplit, the previous best decision tree-based packet classifier.

In Chapter 6, we presented PartitionSort, a packet classifier that partially orders rules by sorting them on their individual fields. It then uses rule list separation to partition the rules into sets that are each totally ordered, allowing for binary searching. This allows it to classify packets 7.2 times faster than Tuple Space Search, the previous choice for OpenFlow packet classification.

Finally, in Chapter 7, we presented TupleMerge, a packet classifier that uses rule list

separation to divide the rules into several hash tables. This improves upon Tuple Space Search by allowing rules with similar, but not identical, tuples to be placed into the same table. This reduces the number of tables compared to Tuple Space Search by 5.4 times. This in turn reduces the search time by 7.43 times.

Further research can be done in this area. ByteCuts currently does not support incremental updates. Adding this ability would allow it to support online packet classification. Additionally, ByteCuts, PartitionSort, and TupleMerge could each be used for different partitions in a rule list separation classifier. Identifying new ways to partition the rules to take advantage of the strengths of each algorithm could allow for further improvements.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] David A. Applegate, Gruia Calinescu, David S. Johnson, Howard Karloff, Katrina Ligett, and Jia Wang. Compressing rectilinear pictures and minimizing access control lists. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1066–1075, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[2] Florin Baboescu and George Varghese. Scalable packet classification. In *In ACM SIGCOMM*, pages 199–210, 2001.

[3] Samuel W. Bent, Daniel D. Sleator, and Robert E. Tarjan. Biased search trees. *SIAM Journal on Computing*, 14(3):545–568, 1985.

[4] James Daly, Alex X. Liu, and Eric Torng. A difference resolution approach to compressing access control lists, 2013.

[5] Pankaj Gupta, , Pankaj Gupta, and Nick Mckeown. Packet classification using hierarchical intelligent cuttings. In *in Hot Interconnects VII*, pages 34–41, 1999.

[6] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields, 1999.

[7] Peng He, Gaogang Xie, Kavé Salamatian, and Laurent Mathy. Meta-algorithms for software-based packet classification. In *Proceedings of the 22nd IEEE International Conference on Network Protocols (ICNP)*, pages 308–319, Washington, DC, USA, 2014. IEEE Computer Society.

[8] Ju Yuan Hsiao, Chuan Yi Tang, and Ruay Shiung Chang. An efficient algorithm for finding a maximum weight 2-independent set on interval graphs. *Infomation Processing Letters*, 43(5):229–235, October 1992.

[9] Kirill Kogan, Sergey Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. Sax-pac (scalable and expressive packet classification). In *Proceedings of the ACM SIGCOMM Conference*, pages 15–26, New York, NY, USA, 2014. ACM.

[10] Maciej Kuzniar, Peter Peresíni, and Dejan Kostic. What you need to know about SDN flow tables. In *Passive and Active Measurement - 16th International Conference, PAM 2015, New York, NY, USA, March 19-20, 2015, Proceedings*, pages 347–359, 2015.

[11] T. V. Lakshman and Dimitrios Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *In ACM SIGCOMM*, pages 203–214, 1998.

[12] Alex X. Liu and Mohamed G. Gouda. Complete redundancy detection in firewalls. In *In Proceedings of 19th Annual IFIP Conference on Data and Applications Security, LNCS 3654, S. Jajodia and D. Wijesekera Ed*, pages 196–209, August 2005.

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

[14] Alex X. Liu, Chad R. Meiners, and Eric Torng. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Transactions on Networking*, 18(2):490–500, 2010.

[15] Alex X. Liu, Eric Torng, and Chad Meiners. Firewall compressor: An algorithm for minimizing firewall policies. In *Proceedings of the 27th Annual IEEE Conference on Computer Communications (INFOCOM)*, pages 176–180, Phoenix, Arizona, April 2008.

[16] Alex X. Liu, Eric Torng, and Chad R. Meiners. Compressing network access control lists. *IEEE Transactions on Parallel and Distributed Systems*, 22:1969–1977, 2011.

[17] Nick Mckeown, Scott Shenker, Tom Anderson, Larry Peterson, Jonathan Turner, Hari Balakrishnan, and Jennifer Rexford. Openflow: Enabling innovation in campus networks.

[18] Chad R. Meiners, Alex X. Liu, and Eric Torng. Topological transformation approaches to optimizing tcam-based packet classification systems.

[19] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[20] Yaxuan Qi, Lianghong Xu, Baohua Yang, Yibo Xue, and Jun Li. Packet classification algorithms: From theory to practice. In *INFOCOM 2009, IEEE*, pages 648–656, 2009.

[21] Matthew J. Saltzman. *Coin-Or: An Open-Source Library for Optimization*, pages 3–32. Springer US, Boston, MA, 2002.

[22] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 213–224, New York, NY, USA, 2003. ACM.

[23] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: An aid to network processing. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for*

Computer Communications, SIGCOMM '05, pages 181–192, New York, NY, USA, 2005. ACM.

[24] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. *SIGCOMM Comput. Commun. Rev.*, 29(4):135–146, August 1999.

[25] V. Srinivasan, George Varghese, Subhash Suri, and Marcel Waldvogel. Fast and scalable layer four switching, 1998.

[26] Xuehong Sun, Sartaj K. Sahni, and Yiqiang Q. Zhao. Packet classification consuming small amount of memory. *IEEE/ACM Trans. Netw.*, 13(5):1135–1145, October 2005.

[27] Subhash Suri, Tuomas Sandholm, and Priyank Ramesh Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 35(4):287–300, 2003.

[28] David E. Taylor. Survey & taxonomy of packet classification techniques. Technical report, ACM COMPUTING SURVEYS, 2004.

[29] David E. Taylor and Jonathan S. Turner. Classbench: a packet classification benchmark. *IEEE/ACM Trans. Netw.*, 15(3):499–511, June 2007.

[30] V. K. Vaishnavi. Multidimensional balanced binary trees. *IEEE Transactions on Computers*, 38(7):968–985, Jul 1989.

[31] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. Efficuts: optimizing packet classification for memory and throughput. *SIGCOMM Comput. Commun. Rev.*, 40(4):207–218, August 2010.