2011

Grouper: A Packet Classification Algorithm Allowing Time-Space Tradeoffs

Joshua Adam Kuhn
University of South Florida, jakuhn2@cse.usf.edu

Grouper: A Packet Classification Algorithm Allowing Time-Space Tradeoffs

by

Josh Kuhn

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Jay Ligatti, Ph.D.
Adriana Iamnitchi, Ph.D.
Ken Christensen, Ph.D.

Date of Approval:
March 22, 2011

Keywords: Network Algorithms, Security, Network Security, Computer Networks, Algorithms

ACKNOWLEDGEMENTS

I would like to acknowledge my major professor, Jay Ligatti, for his guidance and support. I'd also like to thank Chris Gage, Ethan Finkel, and Bhargava Kondaveeti for their assistance with the implementation of Grouper. I would like to thank Girl Talk and Radiohead for their music which I listened to on repeat while implementing Grouper. On the technical side, thanks to Lenovo for making the legendarily solid Thinkpad line of laptops, Richard Stallman for creating Emacs and GCC (and the GPL which we licensed Grouper under), and finally Linus Torvalds for creating Linux. Finally, I would like to thank the NSF, who supported my work with grants CNS-0716343 and CNS-0742736.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

This thesis presents an algorithm for classifying packets according to arbitrary (including noncontiguous) bitmask rules. As its principal novelty, the algorithm is parameterized by the amount of memory available and can customize its data structures to optimize classification time without exceeding the given memory bound. The algorithm thus automatically trades time for space efficiency as needed. The two extremes of this time-space tradeoff (linear search through the rules versus a single table that maps every possible packet to its class number) are special cases of the general algorithm we present. Additional features of the algorithm include its simplicity, its open-source prototype implementation, its good performance even with worst-case rule sets, and its extendability to handle range rules and dynamic updates to rule sets. The contributions of this thesis first appeared in [1].

CHAPTER 1

INTRODUCTION

Packet classifiers are essential components of many network utilities, including routers and security services like firewalls, packet filters, and intrusion-detection systems. Once a network utility classifies a packet (or often just the first in a flow of packets), the utility can perform some actions specific to that class of packets, such as forwarding the packet to a particular destination, dropping the packet, updating some internal state, or logging information about the packet.

A packet classifier inputs a list of rules, each specifying a class of packets matched by that rule. For example, a rule might specify that it matches all TCP packets with any source IP address, any destination IP address of the form 131.247.*.255, source port 118, and any odd-numbered destination port greater than 1023. Given a list of such rules, the classifier typically prepares some data structures that provide a mapping from any incoming packet p to the set of classes—or more commonly, the highest-priority class—that p matches. Thus, the packet classifier's job is to input packets, and for every packet input, output a class number. Typically, by outputting class number n for input packet p, a classifier indicates that the n^{th} rule in its rule list is the first one to match p (classifiers normally assume a final "catch-all" rule to ensure that every input packet matches at least one rule).

1.1 Related Work

Many software algorithms exist for packet classification, and several articles and books survey the field (e.g., [2, 3, 4, 5]). A large number of algorithms take rules defined by range

and prefix patterns since these lend themselves to efficient search. For example, Rovniagin and Wool describe an algorithm they call Geometric Efficient Matching (GEM), which classifies packets based on range patterns [6]. It works by considering each packet dimension as a separate coordinate in a d-dimensional space (where d is the number of packet dimensions the rules are specified over), and partitions the space into areas where different rules match. GEM has a dimension dependent classification time that is $O(d \log n)$, and a worst case space complexity that is $O(n^4)$. Baboescu and Varghese propose an algorithm called Aggregated Bit Vectors (ABVs) in [7] that is an improvement of the Lucent bit vector scheme described by Lakshman and Stiliadis in [8]. These algorithms take prefix and range rules as input and use bitmap intersection of partial matches for each packet field to determine which class a packet belongs to. ABV claims classification rates capable of handling linespeeds equivalent to OC-48, even though classification speed is technically linear in the number of rules.

Qi et al. describe in [9] an algorithm called HyperSplit which is based on mapping the possible packet values into a multi-dimensional space and dividing that space up into regions so that all points (represented by a packet value) in a region match a specific rule. HyperSplit is also defined by range and prefix patterns and gains efficiency over the similar strategy of HiCuts [10]. Both algorithms divide the search space for incoming packets into a heirarchy. Interestingly, classification time for both HiCuts and HyperSplit is $O(d)$, and their space usage is $O(n^d)$ meaning the classification time is dimension dependent (unlike Grouper, as Chapter 3 discusses).

Unfortunately, handling only range/prefix patterns is problematic for rules that could be specified more simply with bitmasks—bitmask patterns cannot in general be translated efficiently into range/prefix patterns. For example, to match IP addresses of the form 131.247.*.255 (e.g., all hosts numbered 255 on any subnet in the 131.247 network) would require 256 range/prefix patterns because the wildcard bits do not appear at the end of the pattern. Similarly, to match all 16-bit port numbers that are odd and greater than 1023 would require only 6 bitmask patterns but 32, 256 range/prefix patterns. In general, a single b-bit bitmask pattern can require

up to 2^{b-1} range/prefix rules in order to match an equivalent set of inputs (specifically when converting a bitmask pattern of the form *1 or *0 into range/prefix patterns). This is not just a theoretical problem; Gupta and McKeown found that about 10% of rules they surveyed in real classifiers contained noncontiguous bitmask patterns [11].

On the other hand, we can convert range/prefix patterns relatively efficiently into bitmask patterns. Prefix patterns are already bitmask patterns, and every range pattern over b bits can be automatically converted into at most $2b-2$ bitmask/prefix patterns (the worst-case conversion occurs for ranges of the form 00...01–11...10) [12]. Based on Gupta and McKeown's survey of practical pattern usage in classification rules, particularly their finding that about 10% of rules used a range pattern but about 90% of all range patterns just specified port numbers greater than 1023 [11], we might expect that converting a range-pattern rule list into a bitmask-pattern rule list would typically inflate the number of rules by about 50%. Also, the worst-case linear inflation when converting range to bitmask patterns is tractable, while the worst-case exponential inflation when converting bitmask to range/prefix patterns is not. Bitmask patterns are therefore more efficiently expressive, in general, than range/prefix patterns alone.

Some software classification solutions can handle noncontiguous-bitmask patterns, such as Recursive Flow Classification (RFC) developed by Gupta and McKeown [11]. RFC is a heuristic that exploits the structure of common rulesets. It uses a feedback mechanism to recursively classify a packet into smaller and smaller sets of possible rules that can match the incoming packet. The authors report, however, that RFC uses a prohibitively large amount of memory for rule sets of more than 6,000 rules. In addition, the memory usage is not tunable and grows exponentially in the number of dimensions the rule set is defined over.

Ternary content-addressable memories (TCAMs) are specialized hardware that can also classify packets using bitmask rules [13, 14]. Currently, they are the de facto industry standard for classfication due to their high throughput because they can compare incoming packets to all rule patterns in parallel [15]. TCAMS are expensive, however, costing up to $250 per Mb of memory [15]. They also consume from 15-30 watts per Mb, leading most of the chips

commercially available to be limited to 128Mb size or less (usually 1-2Mb is common) [15, 16]. Finally, TCAMs are limited in the length of the bitmask rule they can specify. The standard length is 144 bits [15], but this is inadequate for classifying IPv6 headers which have 320 bits at a minimum. We will see that not only can Grouper classify rulesets of hundreds of thousands of rules, it can also handle rules specified over many thousands of bits (i.e., Grouper can classify packets based on rule sets much larger than 128Mb, even on commodity hardware).

Other classification algorithms handle patterns more expressive than bitmasks, including full regular expressions (e.g. EFSAs [17], XFAs [18], and BDDs [19]). However, all algorithms in this category suffer from worst-case exponential (in the number of packet bits classified) memory requirements and/or classification times, due to the arbitrarily complex set of packets specifiable in a single rule.

Another group of classifiers that handle more expressive patterns than bitmasks are the Snort intrusion detection system [20], and the open source firewall iptables [21]. Both of these have very expressive rule languages, which are convenient for describing complicated firewall policies. Unfortunately, both of these implementations rely on a linear search through the rule list to classify each packet. Linear classification time in the number of rules limits their application in environments where high throughput and large rule sets are required because the classifier becomes a bottleneck of the system.

All classification algorithms make some time-space tradeoff between two extremes. At one extreme, a classifier could use no space beyond that of the rule list but have to classify each packet by performing a linear search through the list of rules. In this case, both the space used and packet-classification times are $O(nb/w)$, because each of the n rules may specify b packet bits that have to match the b bits stored in $O(b/w)$ machine words (where w is the word size in bits; we analyze space usage and classification time in terms of memory words stored/accessed). At the other extreme, a classifier could maintain a single table that maps each of the 2^b possible input packets to its class number. In this case, the space required for storing 2^b entries of class numbers each having size $O((\lg n)/w)$ is $O((2^b \lg n)/w)$, while the

classification time is only $O((\lg n)/w)$. Linear search is space efficient but runtime inefficient, while a single table is runtime efficient but space inefficient.

1.2 Contributions

This thesis presents an algorithm called Grouper (described in detail in Chapter 2) for classifying packets according to bitmask rules. The algorithm partitions the bits being classified into approximately equal-sized groups and uses each value of grouped bits in a packet to look up a bitmap of rules matching that group value. It computes the set of rules matching any packet by intersecting the sets of rules matching each of that packet's grouped bits. Thus, Grouper uses the common technique of intersecting sets of matched rules [8, 11, 7], but unlike any rule-set-intersection algorithms we are aware of, Grouper classifies according to arbitrary (including noncontiguous) bitmask rules while exhibiting good performance even on large rule sets (having many thousands of rules).

By controlling the sizes of bit groupings, Grouper can control the amount of memory needed for its bitmap-lookup tables; larger group sizes imply larger amounts of memory consumed but faster classification times. Thus, the algorithm can customize its data structures to optimize classification time without exceeding a given memory bound. This ability to automatically trade time for space efficiency is the algorithm's principal novelty. Besides automatically trading time for space and classifying according to arbitrary bitmask policies, Grouper features: simplicity, an open-source prototype implementation [22], good performance even with worst-case rule sets, and extendability to handle range rules and dynamic updates to rule sets.

As described in Chapter 3, the experiments performed have shown that Grouper is capable, when implemented in software on a commodity laptop using about 2GB of memory, of classifying entire 320-bit IPv6 headers into one of 1,000 (respectively 100,000) randomly generated classes at 579,397 (16,774) pps. When classifying according to only 100 randomly generated rules, but with each rule specifying a bitmask over a full 12,000 bits of Ethernet payload, we

observed classification throughputs of 25,271 pps (i.e., several hundred Mbps, again in a software implementation). Grouper can classify hundreds/thousands of packet bits efficiently, in part because it operates independently of the number of packet fields/dimensions being classified.

CHAPTER 2

THE GROUPER ALGORITHM

Grouper uses t lookup tables to classify b packet bits according to n rules (see Figure 2.1). Each lookup table maps either $\lfloor b/t \rfloor$ or $\lceil b/t \rceil$ (referred to as "floor" and "ceiling" groups respectively) of the b packet bits to an n-length bitmap indicating which of the n rules match those $\lfloor b/t \rfloor$ or $\lceil b/t \rceil$ packet bits. We say the $\lfloor b/t \rfloor$ or $\lceil b/t \rceil$ bits used to index a table are *grouped* together, with a group size of $\lfloor b/t \rfloor$ or $\lceil b/t \rceil$ bits. Every table maps a group of bits to an n-length bitmap (see Figure 2.2).

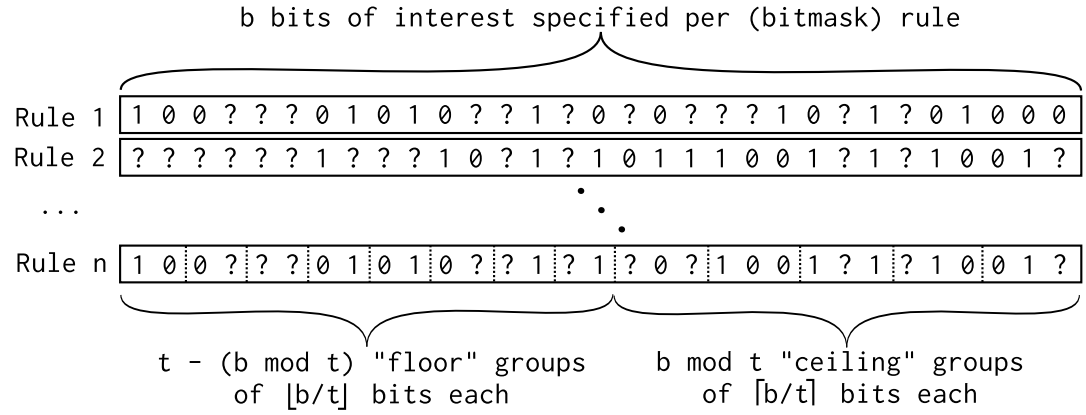

Figure 2.1. Layout of the rule sets Grouper operates on

Grouper uses the lookup tables as follows. Given b bits of an input packet to classify, Grouper divides those b bits into t groups and uses the values of the bits in each group to index into a table to look up the n-length bitmap of rules matching that group of bits. By intersecting (i.e., bitwise ANDing) all bitmaps of rules matching every group, Grouper ends up with an n-length bitmap of rules matching the entire input packet. The first set bit in that final bitmap

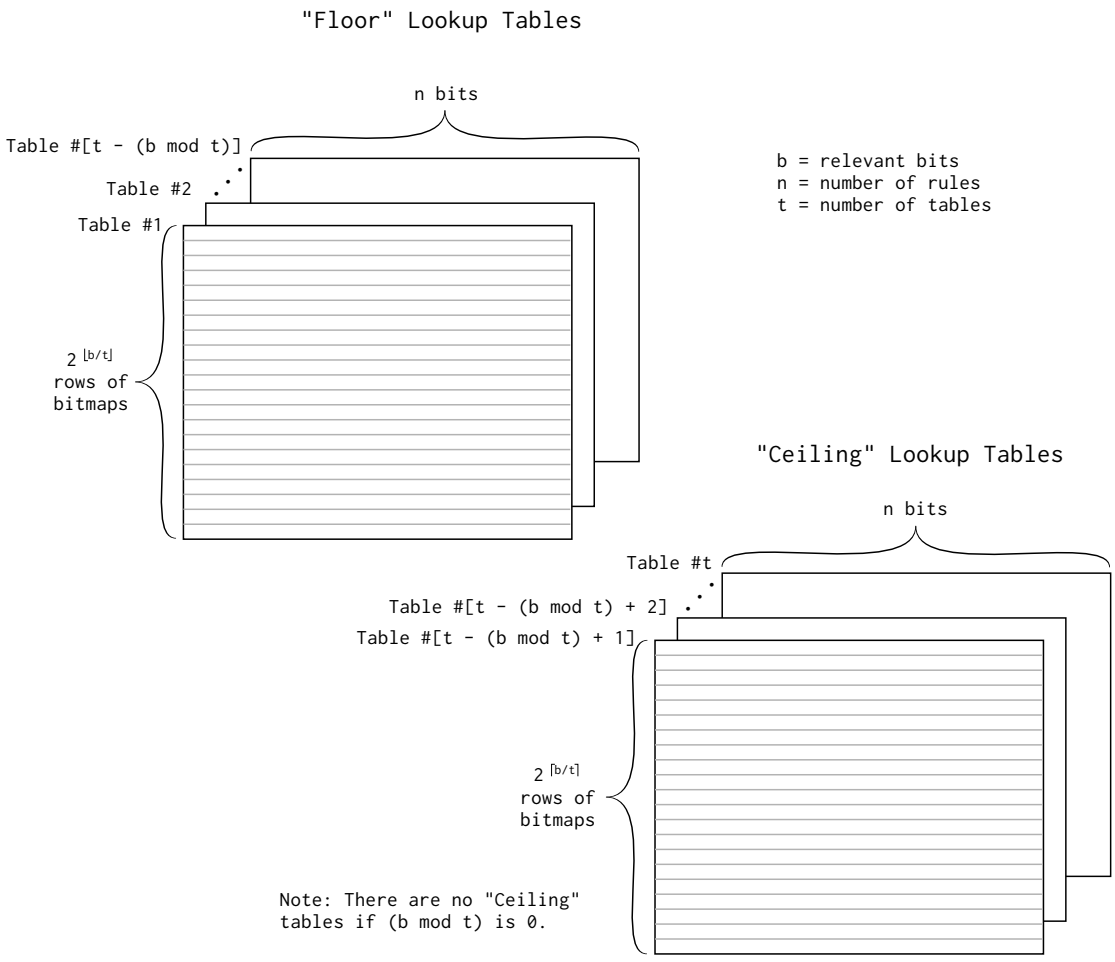

Figure 2.2. Diagram of the Grouper classification tables

indicates the lowest-numbered (highest-priority) rule matching the original b input bits. The algorithm in Figure 2.4 formally describes how Grouper classifies packets.

Because Grouper does not guarantee that it will group any two particular bits together, it may form groups of arbitrary bits from the b-bit input. One consequence of this arbitrariness in bit groupings is that Grouper operates independently of packet fields/dimensions; the algorithm simply views its input as b packet bits, regardless of higher-level categorizing of those bits into fields. A second consequence of grouping arbitrary bits together is that other packet bits cannot influence what a particular bit value matches; it is this constraint that limits Grouper (in its basic version) to classifying according to bitmask rules.

Grouper does however guarantee that it will partition the b packet bits into t groups having as equal of size as possible (i.e., either $\lfloor b/t \rfloor$ or $\lceil b/t \rceil$ bits). Evening out the group sizes in this way evens out the number of bits used to index each table, thus preventing (1) space inefficiencies that arise with disproportionately large tables and (2) time inefficiencies that arise with disproportionately small tables. Since the bitmap is the same length for each table (it is proportional to n, not to the number of bits that index the table), disproportionately small tables incur the same runtime hit as a large table, but provide less information for classification.

The algorithm in Figure 2.3 shows how Grouper constructs its tables.

Input : The number of tables to build, t
Input : The number of relevant bits in a rule, b
Input : The total number of rules, n
Input : Rules, a series of bitmask rules indexable by bits. (Example: Figure 2.1)
Output: A series of tables that can be used by the classification algorithm in Fig. 2.4
begin
  Tables ←a zeroed series of tables with the structure shown in Fig. 2.2
  for i $= 1 \rightarrow$ t $-$ (b $\mod$ t) **do**
    L $\leftarrow$ (i $-$ 1) $\cdot \lfloor$ b/t $\rfloor + 1$
    H $\leftarrow$ L $+ \lfloor$ b/t $\rfloor - 1$
    for j $= 0 \rightarrow 2^{\lfloor b/t \rfloor} - 1$ **do**
      for k $= 1 \rightarrow$ n **do**
        if *bits* L *through* H *of* Rules[k] *match* j **then**
          Set bit k in table i, row j in Tables
  offset $\leftarrow$ t $-$ (b $\mod$ t)
  for i $=$ t $-$ (b $\mod$ t) $\rightarrow$ t **do**
    L $\leftarrow$ (i $-$ offset $- 1$) $\cdot \lceil$ b/t $\rceil +$ offset $\cdot \lfloor$ b/t $\rfloor + 1$
    H $\leftarrow$ L $+ \lceil$ b/t $\rceil + 1$
    for j $= 0 \rightarrow 2^{\lceil b/t \rceil} - 1$ **do**
      for k $= 1 \rightarrow$ n **do**
        if *bits* L *through* H *of* Rules[k] *match* j **then**
          Set bit k in table i, row j in Tables
  return Tables

Figure 2.3. Basic Algorithm for table building

Input : A set of Grouper Tables created by the algorithm in Fig. 2.3
Input : The number of Grouper tables, t
Input : The number of relevant bits in a rule, b
Input : The total number of rules, n
Input : A bitmap p of length b, the incoming packet bits to be classified
Output: The class number the incoming packet p belongs to
begin
  OutVector $\leftarrow$ a zeroed bitmap of length n bits
  for $i = 1 \rightarrow$ t **do**
    if i $\leq$ t $-$ (b $\mod$ t) **then**
      s $\leftarrow \lfloor$b/t$\rfloor$
      i_offset $\leftarrow 0$
      b_offset $\leftarrow 0$
    else
      s $\leftarrow \lceil$b/t$\rceil$
      i_offset $\leftarrow$ t $-$ (b $\mod$ t)
      b_offset $\leftarrow$ i_offset $\cdot \lfloor$b/t$\rfloor$
    L $\leftarrow$ (i $-$ i_offset $- 1) \cdot$ s $+$ b_offset $+ 1$
    H $\leftarrow$ L $+$ s $- 1$
    rownum $\leftarrow$ bits L through H (inclusive) of p
    OutVector $\leftarrow$ OutVector bitwise ANDed with row rownum, table i in Tables
  return Position of the first set bit in OutVector or 0 if no bit is set

Figure 2.4. Algorithm for packet classification

2.1 Possibilities for the Number of Lookup Tables

As a special case, when $t = 1$, Grouper does not have to perform any bitmap intersections, so its one lookup table, indexed by all b packet bits, can be optimized to store not bitmaps but the actual class numbers matching all possible b-bit values. Hence, the special case of $t = 1$ corresponds to one extreme in the time-space tradeoff of packet classification, in which a single table maps all possible b-bit values to their class numbers.

As another special case, when $t = b$, every lookup table maps a single bit of the input packet to a bitmap indicating which rules match that bit value. In this case Grouper classifies by iterating through every bit of input and intersecting the bitmap for each input bit to determine which rules match all input bits. This approach is conceptually the same as a linear-search classification algorithm: both approaches iterate through all possible pairings of input bits and

rule numbers to find which rule numbers match all the input bits; both approaches classify in time $O(bn/w)$ using $O(bn/w)$ space. Hence, the special case of $t = b$ corresponds to the other extreme in the time-space tradeoff of packet classification in which the algorithm performs a linear search.

In general, setting t to a lower value causes Grouper to use more space but classify packets more quickly (fewer tables have to be queried and fewer bitmaps have to be intersected). The special cases of $t = 1$ and $t = b$ correspond to extremes of the time-space tradeoff in packet classification. However, it turns out that it never makes sense to set $t > \lceil b/2 \rceil$ because any such t value saves no space compared to setting $t = \lceil b/2 \rceil$. To see why, consider the hypothetically most space-saving setting of t to b; in this case each of the b tables stores two n-length bitmaps. We can always replace two such tables (consuming a total of $4\lceil n/w \rceil$ space) with a single table that maps 2 bits of the input packet to four possible n-length bitmaps (also consuming a total of $4\lceil n/w \rceil$ space). Thus, it only makes sense to use Grouper with t values ranging from 1 (corresponding to the single-lookup-table algorithm) to $\lceil b/2 \rceil$ (corresponding to the linear-search algorithm).

2.2 Memory Use, Table-build Time, and Classification Time

Grouper uses t tables, each having $O(2^{b/t})$ entries, with each entry being an n-length bitmap consuming $O(n/w)$ machine words. The total memory words used is therefore $O((2^{b/t}tn)/w)$, where again, t can range from 1 to $\lceil b/2 \rceil$. More precisely, the following equation gives the number of bits m required to store Grouper's tables.

$$m = \begin{cases} (t - (b \bmod t)) \cdot 2^{\lfloor b/t \rfloor} \cdot n + (b \bmod t) \cdot 2^{\lceil b/t \rceil} \cdot n & \text{if } 2 \leq t \leq \lceil b/2 \rceil \\ 2^b \cdot \lceil \lg n \rceil & \text{if } t = 1 \end{cases} \tag{2.1}$$

Equation 2.1 partitions the b input bits into t groups such that every group has as uniform as possible of a size: $b \bmod t$ groups will contain $\lceil b/t \rceil$ bits, while $t - (b \bmod t)$ groups will contain $\lfloor b/t \rfloor$ bits. We consequently have $b \bmod t$ tables of $2^{\lceil b/t \rceil}$ entries and $t - (b \bmod t)$ tables

of $2^{\lfloor b/t \rfloor}$ entries. Building the full lookup tables from scratch may require time proportional to their size, with Grouper iterating over and setting every table entry.

Grouper's space requirements are exponential in the number of bits in each group, which is different from most other comparable classification algorithms whose space usage is defined in terms of the (fixed) number of packet dimensions its rule set is defined over (usually denoted d). For example, both the cross-producting technique [12], and RFC [11] use memory that is $O(n^d)$, where N is the number of rules [4].

Classification time for Grouper is $O(tn/w)$ because it queries every one of the t tables to obtain a bitmap consuming $O(n/w)$ memory words, and as Grouper fetches each of those bitmaps, they get intersected with any previously fetched bitmaps. Although this classification time is linear in n (the number of rules), Grouper, like other bitmap-intersection algorithms, benefits from (1) storing rule information in bitmaps to divide the n factor in the classification time by the word size, and (2) spatial locality of bits fetched in bitmaps, resulting in good cache performance. This is in contrast to some other packet classifiers like Snort and iptables whose best case classification time is linear in the number of rules (i.e. $O(n)$ [20, 21]).

2.3 Optimizing Classification Time without Exceeding a Given Memory Bound

Minimizing Grouper's $O(tn/w)$ classification time requires minimizing t (number of tables) and n (number of rules) and maximizing w (word size). The rule set dictates n and hardware dictates w, making these parameters beyond Grouper's control. Grouper can however minimize t such that its tables fit within a given memory constraint. To take advantage of this ability, we parameterize Grouper by not only a classification policy, but also a memory constraint; Grouper will automatically (during table preprocessing) trade time for space efficiency to make its lookup tables as runtime efficient as possible while obeying the given memory constraint.

Equation 2.1 already shows how to calculate m (the number of bits needed for Grouper's lookup tables) when given t, b, and n. To calculate a minimum t when given a maximum m

and a classification policy (which determines the values of b and n), Grouper simply (1) checks whether a single lookup table consumes less memory than the given m value; if not then (2) performs a binary search between all possible t values (from 2 to $\lceil b/2 \rceil$) to find the smallest one that, when plugged into Equation 2.1 with the given b and n values (and rounding n to the next multiple of 8), produces a memory requirement no greater than the given maximum m value (see Figure 2.5). This binary-search algorithm produces the optimal t in $O(\lg b)$ time.

```
Input  : Maximum memory allowed in bits, m
Input  : Number of rules, n
Input  : Number of relevant bits, b
Output: Minimum number of tables that fit within the given memory bound, or -1 if
        there is no possible value of t that respects this bound
begin
    if m < 2 · n · b or n < 1 or b < 1 then
        return −1
    if m ≥ log n · 2^b then
        return 1
    low ← ⌈b/2⌉
    high ← 1
    while (low-high) > 1 do
        mid ← (low + high)/2
        FloorTablesMem ← (mid − (b mod mid)) · 2^⌊b/mid⌋ · n
        CeilingTablesMem ← (b mod mid) · 2^⌈b/mid⌉ · n
        memNeededForTables ← FloorTablesMem + CeilingTablesMem
        if m < memNeededForTables then
            high ← mid
        else
            low ← mid
    return low
```

Figure 2.5. Algorithm to determine the minimum tables for a given memory bound

2.4 A Simple Example

In order to better understand how Grouper works, it can be helpful to have an example. Consider the rule set in Figure 2.6. For simplicity, this rule set has only two rules of seven bits each ($n = 2$, $b = 7$). In addition, the example limits the algorithm to using only 24 bits for

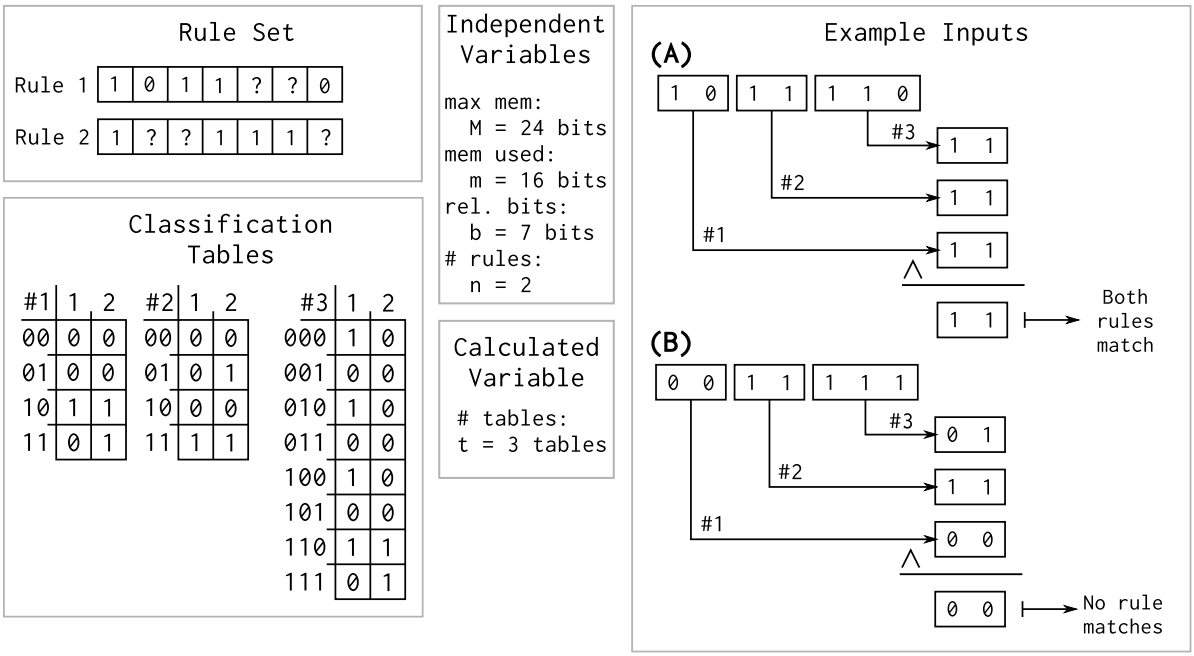

Figure 2.6. A simple example of how Grouper works

the classification tables ($M = 24$). With a number of relevant bits, b, this small, there are only 3 values for t that make sense: 1, 2 or 3 tables (see Section 2.1). If we compute the memory usage with equation 2.1 using these parameters , we get memory requirements of 256, 48 and 16 bits, respectively. Since 16 bits is the only result that fits within the memory limit, $t = 3$ (with so few possibilities for tables, it is easy to calculate all of their memory requirements, but the algorithm in Figure 2.5 performs this calculation faster for larger values of b).

Next, Grouper builds the actual classification tables as shown in Figure 2.6. In the diagram, the numbers along the top of the tables represent the rule numbers. The numbers along the left side of the tables represent the row number expressed in binary (for ease of comparing with the bitmask rules).

Figure 2.6 contains two example inputs, (A) `1011110` and (B) `0011111`. Grouper breaks the input into three groups, two of 2 bits and one of 3 bits, and retrieves the bitmap from the corresponding row of the corresponding classification table. Then it performs a bitwise AND. In (A), both rules match the input, but Grouper returns only the highest priority rule (the least

significant set bit in the bitmap), so rule 1 would be output. In (B), neither rule matches the input, so Grouper would output the default 0.

CHAPTER 3

EMPIRICAL ANALYSIS

3.1 Motivation for Implementation

While we can determine the theoretical performance of Grouper, it's helpful gauge the real world performance of the algorithm in order to place it in some context. For example, it is useful to compare the throughput of the algorithm to the maximum throughputs of various common network capacities such as 10 gigabit Ethernet. In addition, we also wanted to measure how Grouper behaves both under realistic conditions and "extreme stress" conditions both with rule sets concerned with thousands of bits and with rule sets containing very large numbers of rules.

Finally, while the Grouper algorithm should scale smoothly in performance when changing the amount of memory available to it, in practice, on real machines there are numerous complicating factors such as virtual memory, multi-level processor caches, branch prediction, and OS context switches that can decrease the reliability of Grouper's performance. In our implementation and testing, we attempt to minimize or account for these effects as much as possible, but it's not possible to eliminate them entirely.

3.2 Implementation

We implemented a prototype of Grouper in 1093 lines of C code. The source code and benchmarking scripts are available online [22]. We compiled the program for the x86-64 architecture, which adheres to the AMD64 specification and includes 16 128-bit multimedia registers. When compiled with `gcc`'s `-O3` option, our prototype performs bitmap intersections in

these 128-bit registers. Our prototype also mitigates the inefficiency of addressing individual bits on a byte-addressable machine by padding all bitmaps to coincide with byte boundaries (hence using $\lceil b/8 \rceil \cdot 8$ bits instead of n bits as the bitmap length).

Our implementation multithreads the table-build (preprocessing) operation to speed this operation up in proportion to the number of processor cores. Like all bitmap-intersection algorithms, we believe Grouper's packet-classification operations are amenable to parallelization (or pipelining). We briefly experimented with performing classifications in two threads (one for each of the two processor cores on our test machine) but found the context-switching costs outweighed the benefits of concurrency in this case, so we reverted to a single-thread implementation.

3.3 Experimental Setup

We tested Grouper's performance on a Dell Latitude D630 with 2GHz Intel Core 2 Duo processors, running a minimal version of Arch Linux. Although the laptop had 4GB of memory, we limited the memory used by Grouper to about 2GB to prevent Grouper from contending with any system software for memory.

Our experiments had three independent variables (b, n, and t) and two dependent variables (throughput and table-build times). The b values tested were: 32 (corresponding to classification based on and IPv4 address or source plus destination port), 104 (corresponding to classification based on an 8-bit protocol number, source and destination port numbers at 16 bits each, and source and destination IPv4 addresses), 320 (corresponding to classification based on an entire fixed portion of an IPv6 header), and 12,000 (corresponding to classification based on the entire contents of a maximum-sized Ethernet v2 payload). The n values tested were: 100, 1K, 10K, 100K, and 1M (in this thesis, K, M, and G refer to 10^3, 10^6, and 10^9). The t values tested were: every value from the maximum of $b/2$ tables, down to the minimum number of tables possible without exceeding about 2GB of memory (the minimum t over all tests was 2, which was possible with $b = 32$ and $n \leq$100K). The one exception to this universe of indepen-

dent variables is that we could not test Grouper's performance classifying 12K bits according to 1M rules because doing so would require about 3GB of memory, even using the maximum t value possible. Hence, we report no results for this case of b=12K and n=1M.

For every combination of b, n, and t values, we measured throughput and table-build time for a randomly generated rule set. We measured the throughput by creating a file of 500K random b-length packets (the exception being that we only used 10K random packets when b was 12K), starting a real-time timer just before the first b bits were read from that file, having Grouper input and classify b-length packets one at a time from the file, and stopping the timer just after Grouper finished classifying all packets in the file; this process produced a pps measurement based on real (including file I/O) time. In the following section, we calculated all measurements reported in terms of bps from the original pps measurement using a fixed packet size of 12K bits. In addition, all the table-build times are wall clock time. We performed all tests three times (the data points in the graphs here are averages of the three trials).

3.4 Results

The graphs in Figures 3.1–3.2 summarize our experimental results. These figures present throughputs and table-build times for given values of n and b. The throughputs are represented in both absolute bits per second on the right axes, and packets per second on the left axes. For each combination of n and b, the graphs display two points: (1) an upper point corresponding to the throughput (or table-build time) with Grouper using the maximum amount of memory available to it, up to about 2GB, and (2) a lower point corresponding to the throughput (or table-build time) with Grouper using the minimum amount of memory possible (i.e., with $t = b/2$). For example, with all 2GB of memory available, Grouper's throughput was 25K pps (300 Mbps) for b=12K and n=100, 140K pps (1.68 Gbps) for b=320 and n=10K, and 1.1M pps (13.2 Gbps) for b=104 and n=1K. Figure 3.1 shows that our prototype software implementation performs well, particularly given that rule sets often have fewer than 1K rules [11, 23]. Figures 3.1–3.2 (whose graphs have log-scale axes) also illustrate that improving classification

throughput by a constant factor requires exponentially greater memory, implying exponentially greater table-build times.

Figures 3.3–3.5 fix the number of rules at 1K, 10K, and 100K, so we can view the classification throughputs in terms of memory consumption. The high throughput when t=2 led us to break the y-axes in these graphs. Also, the throughput dips that occur in Figure 3.3, even as the amount of memory used increases, are due to our test machine's 4MB L2 cache size.

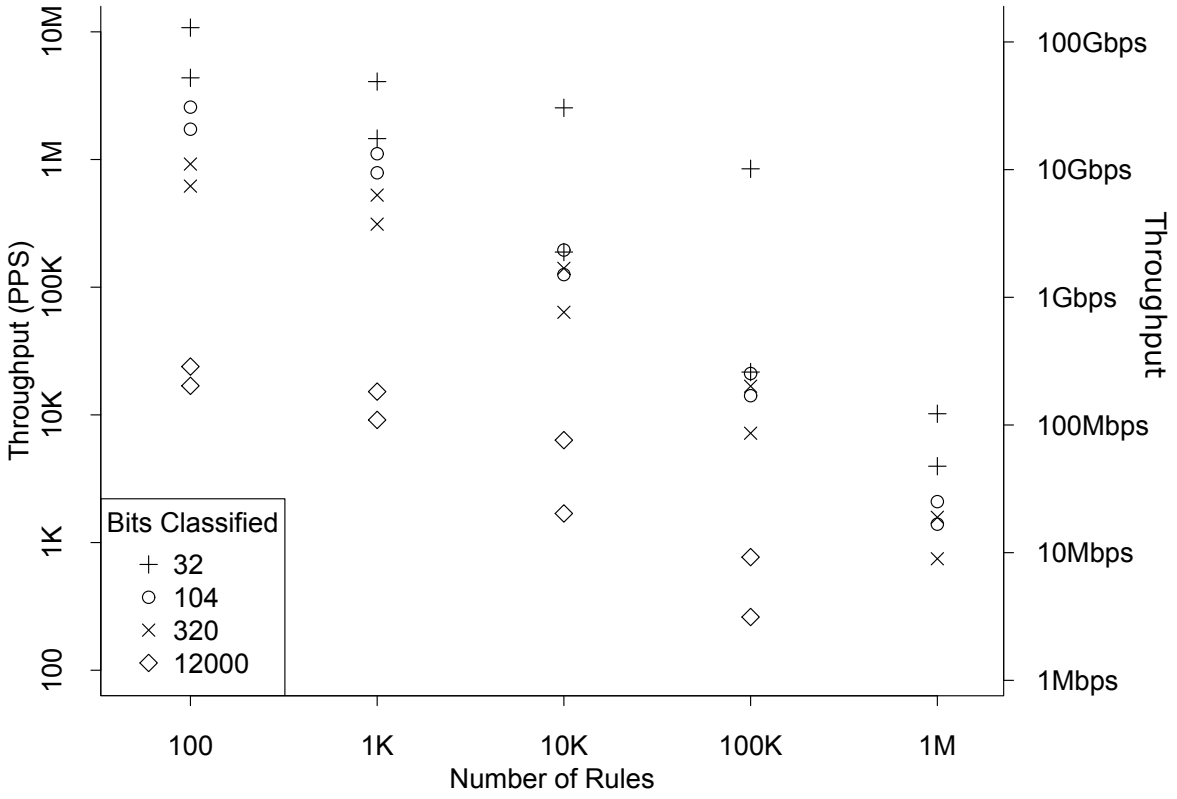

Figure 3.1. Maximum and minimum classifier throughputs

Finally, Figures 3.6–3.7 depict Grouper's performance in a couple of extreme cases where Grouper is classifying either a large number of bits or is classifying using a very large ruleset. These graphs illustrate the inverse relationship between throughput (y-axis) and number of tables (x-axis), which results from Grouper's $O(tn/w)$ classification time per packet.

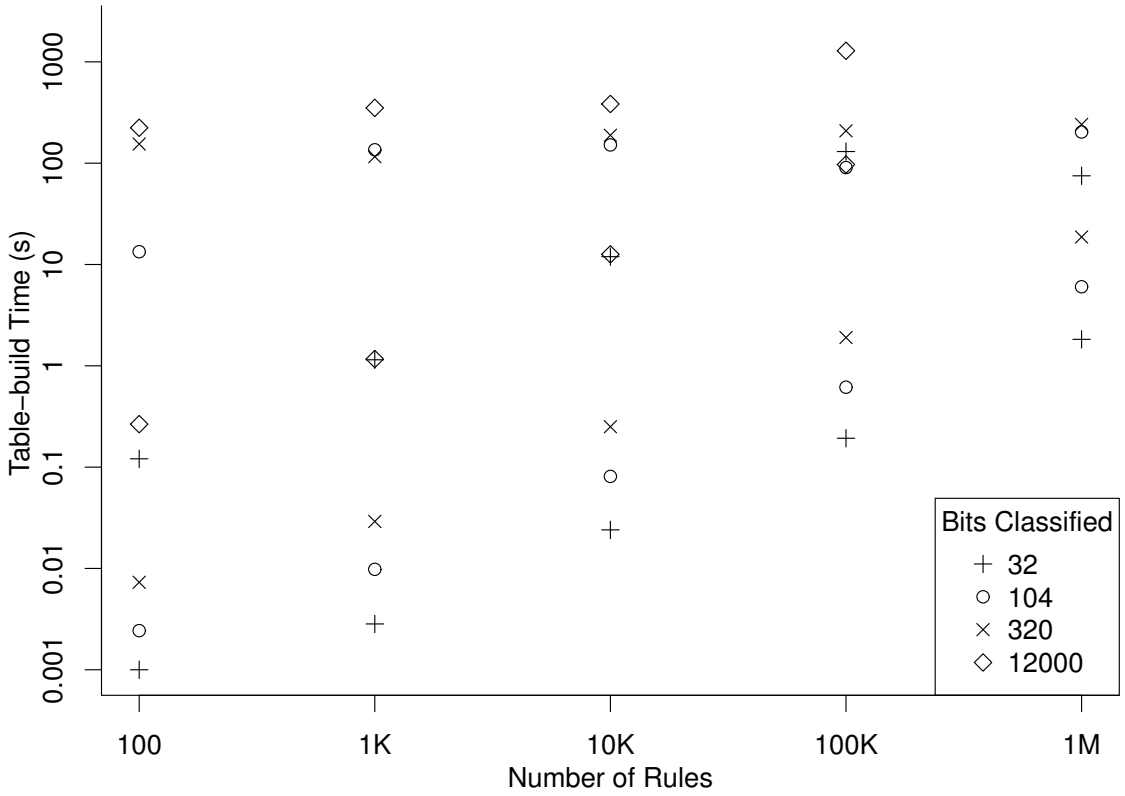

Figure 3.2. Maximum and minimum table-build times

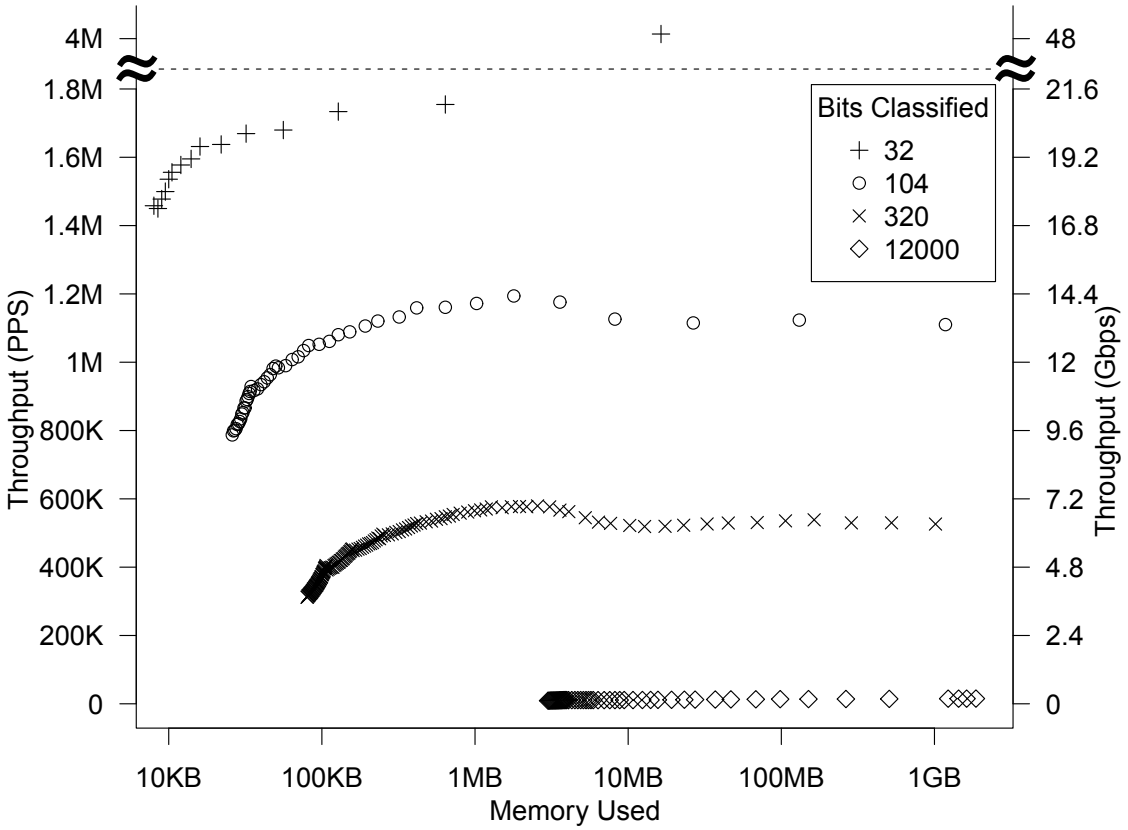

Figure 3.3. Throughputs for 1,000 rules

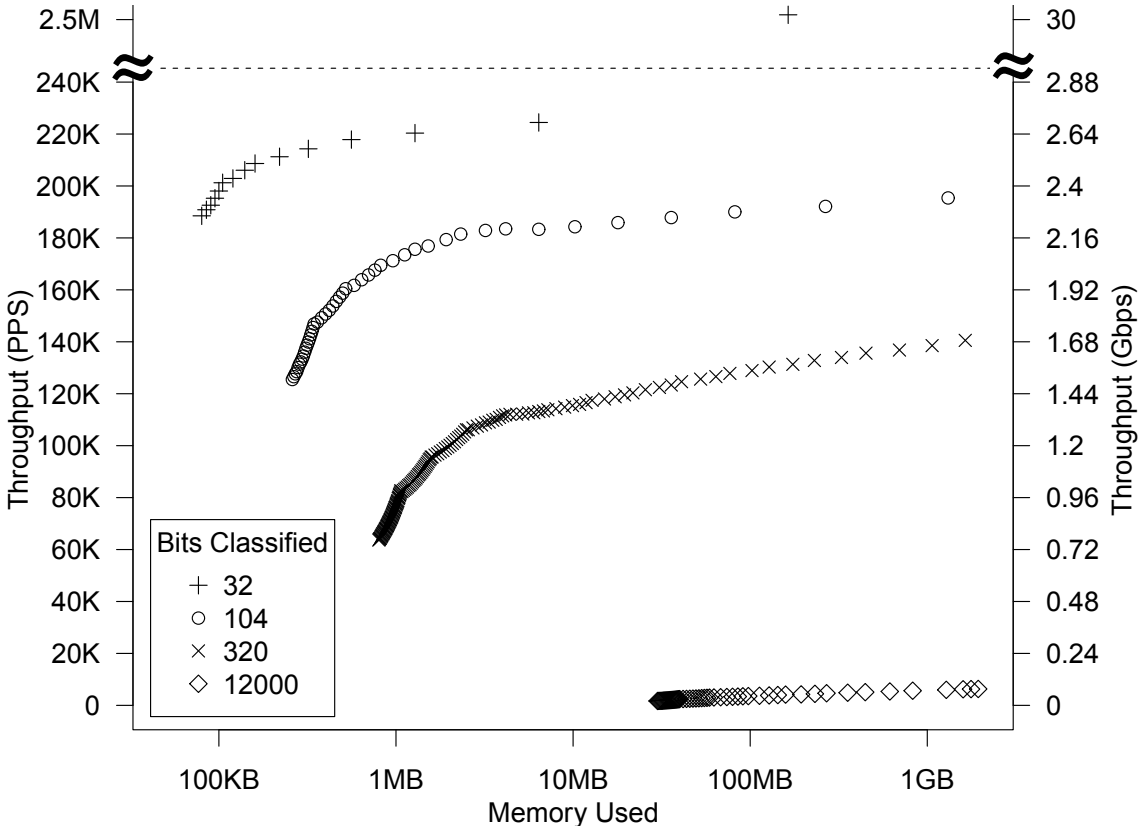

Figure 3.4. Throughputs for 10,000 rules

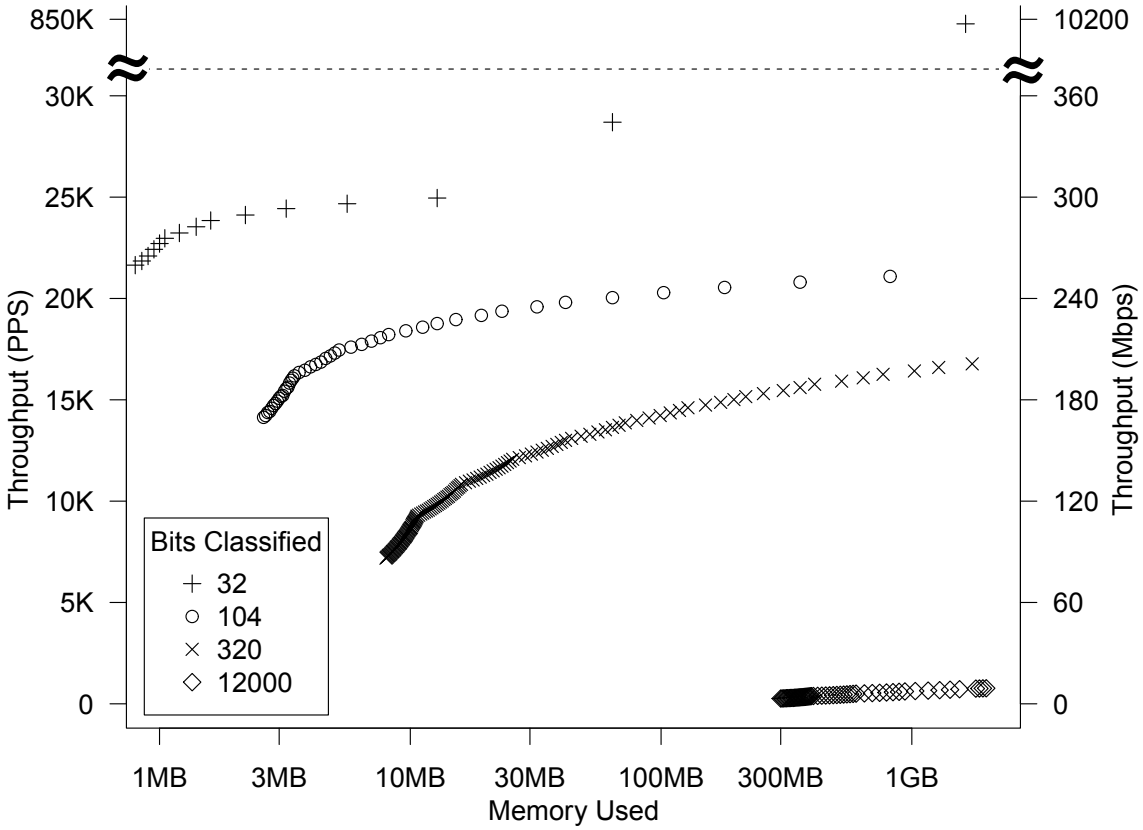

Figure 3.5. Throughputs for 100,000 rules

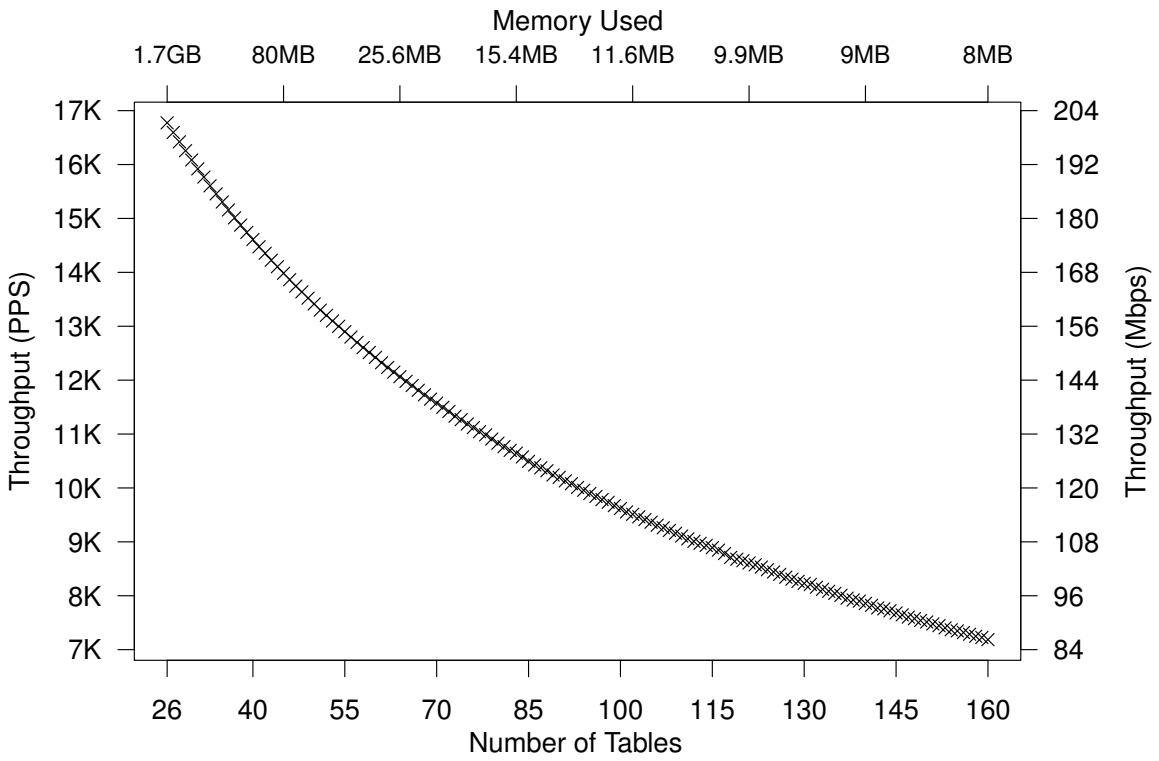

Figure 3.6. Throughputs for 320 bits classified, with 100,000 rules

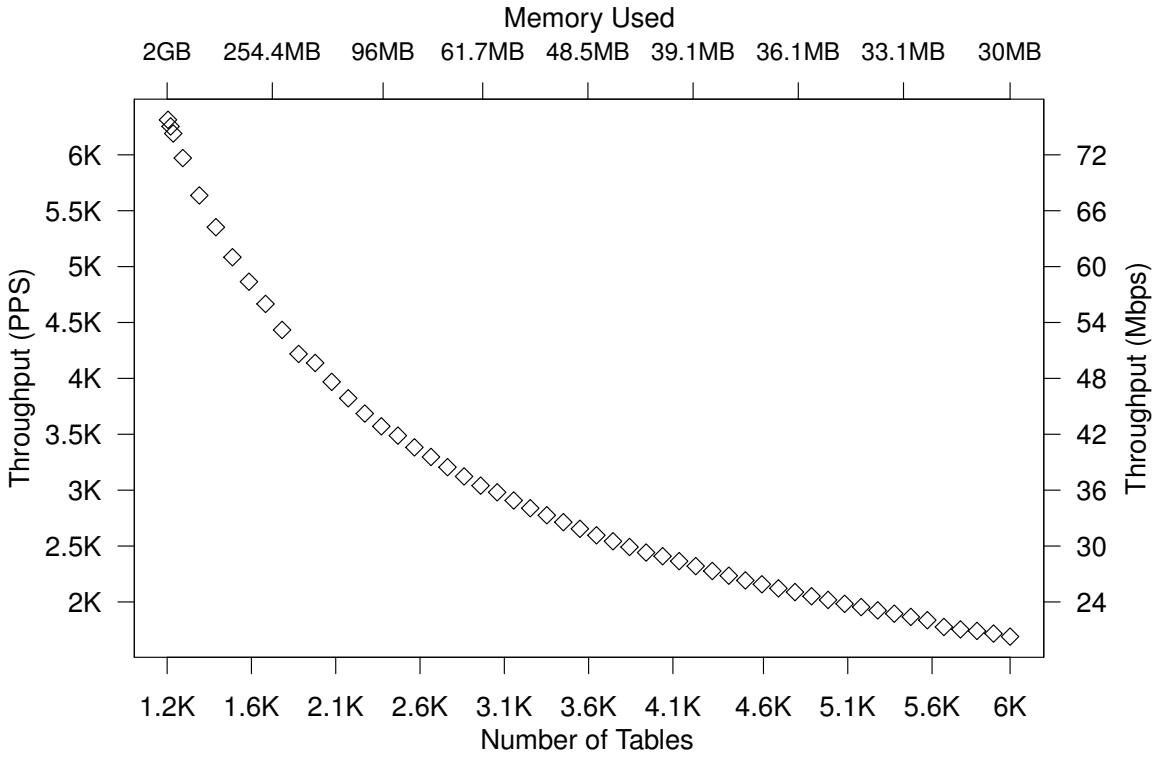

Figure 3.7. Throughputs for 12,000 bits classified, with 10,000 rules

CHAPTER 4

FINAL REMARKS

4.1 Discussion of Extensions

We are considering two extensions to Grouper. The first is adding the ability to specify ranges in the rule sets. Several papers (e.g. [24, 25]) discuss algorithms to expand a classification rule specified as a range into a series of bitmask rules (usually in the context of TCAMs). In the worst case, a single range will require a number of bitmask rules proportional to the number of bits over which the rule is specified (e.g. sixteen bitmask rules for a range over sixteen bits of the input). Unfortunately while this is a nice upper bound for a single range, many common classification rules require specifying more than one range per rule. For example, this happens when specifying one range for the source port and one range for the destination port. In this case, we require the Cartesian product of the individual range expansions in order to correctly match all possible combinations of matches. This means that range expansion is exponential in the number of ranges specified in the rule.

Fortunately, Grouper offers an additional way to handle range rules. Instead of converting the range rules directly to a series of bitmask rules, we simply group all bits of each range into their own table. We can then build the tables by setting the entries in the table that fall within the range's bounds. This loses the flexibility of being able to make the group sizes very small (since all the bits in each the range must be in the same group), but has the advantage that now the memory required does not scale exponentially with the number of ranges specified. Which strategy results in less memory usage depends on the details of the rule set, so ideally Grouper can compare the memory costs before building its tables and adjust its strategy accordingly.

The second extension would handle dynamic updates to rule sets, for those cases where rebuilding lookup tables from scratch takes too long (cf. Figure 3.2). To handle dynamic rule deletions, Grouper could identify rules with internal numbers, which may differ from the externally defined class numbers. For example, after deleting Rule 0, Grouper could continue to identify the new Rule 0 internally as Rule 1. This decoupling of internal rule numbers from external class numbers, combined with a few additional data structures (an extra bitmap with 0s in positions of deleted internal rules and a map from internal to external numbers), enables Grouper to process rule deletions efficiently. To handle dynamic rule additions, Grouper could allocate larger bitmaps than it needs initially. If this extra space ever gets exhausted, Grouper could use another thread to rebuild the tables without taking the classification thread offline.

4.2 Summary

This thesis has presented Grouper, an algorithm for classifying packets according to arbitrary-bitmask rules. Grouper is parameterized by the amount of memory available for its lookup tables and automatically trades time for space efficiency as needed to fit within a given memory bound. Experiments with Grouper's open-source prototype implementation on a commodity laptop have demonstrated its good performance, particularly when classifying based on large numbers of packet bits (e.g., 300 Mbps with b=12K and n=100, 1.68 Gbps with b=320 and n=10K, and 13.2 Gbps with b=104 and n=1K). Because the bitmasks used in a rule set have no effect on Grouper's performance, our experimental results on randomly generated rule sets demonstrate Grouper's performance on worst-case rule sets. In addition, Grouper lends itself to being extended to handle range rules. Given its flexibility and performance, Grouper is a compelling packet-classification algorithm.

LIST OF REFERENCES

[1] Jay Ligatti, Josh Kuhn, and Chris Gage. A packet-classification algorithm for arbitrary bitmask rules, with automatic time-space tradeoffs. In *Proceedings of the International Conference on Computer Communication Networks (ICCCN)*, August 2010.

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

[3] Sataj Sahni, Kun Suk Kim, and Haibin Lu. IP router tables. In Dinesh Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 48. Chapman & Hall/CRC, 2005.

[4] Pankaj Gupta. Multi-dimensional packet classification. In Dinesh Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 49. Chapman & Hall/CRC, 2005.

[5] Deepankar Medhi and Karthikeyan Ramasamy. *Network Routing: Algorithms, Protocols, and Architectures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[6] D. Rovniagin and A. Wool. The geometric efficient matching algorithm for firewalls. In *Proceedings of the IEEE Convention of Electrical and Electronics Engineers in Israel*, 2004.

[7] Florin Baboescu and George Varghese. Scalable packet classification. *IEEE/ACM Trans. Netw.*, 13(1):2–14, 2005.

[8] T. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *SIGCOMM Comput. Commun. Rev.*, 28(4):203–214, 1998.

[9] Yaxuan Qi, Lianghong Xu, Baohua Yang, Yibo Xue, and Jun Li. Packet classification algorithms: From theory to practice. In *Proceedings of Infocom*, 2009.

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

[11] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *Proceedings of SIGCOMM*, 1999.

[12] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. *SIGCOMM Comput. Commun. Rev.*, 28(4):191–202, 1998.

[13] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for advanced packet classification with ternary CAMs. In *Proceedings of SIGCOMM*, 2005.

[14] Chad R. Meiners, Alex X. Liu, and Eric Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. In *Proceedings of the IEEE International Conference on Network Protocols*, pages 93–102, October 2009.

[15] Chad R. Meiners, Alex X. Liu, and Eric Torng. Tcam razor: A systematic approach towards minimizing packet classifiers in tcams. *Network Protocols, IEEE International Conference on*, 0:266–275, 2007.

[16] Inc Cisco Systems. Cisco catalyst 6500 series switch. http://www.cisco.com/en/US/products/hw/switches/ps708/products_white_paper09186a00800c9470.shtml#wp39459.

[17] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the USENIX Security Symposium*, 1999.

[18] Randy Smith, Cristian Estan, and Somesh Jha. XFA: Faster signature matching with extended automata. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 187–201, 2008.

[19] Scott Hazelhurst, Adi Attar, and Raymond Sinnappan. Algorithms for improving the dependability of firewall and filter rule lists. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 576–585, 2000.

[20] Sourcefire, Inc. *Snort*. http://www.snort.org/.

[21] Netfilter Core Team. Iptables webpage. http://www.netfilter.org/projects/iptables/.

[22] Josh Kuhn, Jay Ligatti, and Chris Gage. The grouper webpage. http://www.cse.usf.edu/ ligatti/projects/grouper/.

[23] Avishai Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67, 2004.

[24] Baruch Schieber, Daniel Geist, and Ayal Zaks. Computing the minimum dnf representation of boolean functions defined by intervals. *Discrete Applied Mathematics*, 149(1-3):154 – 173, 2005. Boolean and Pseudo-Boolean Functions.

[25] O. Rottenstreich and I. Keslassy. Worst-case tcam rule expansion. In *INFOCOM, 2010 Proceedings IEEE*, pages 1 –5, March 2010.