A Packet-classification Algorithm for Arbitrary Bitmask Rules, with Automatic Time-space Tradeoffs

Jay Ligatti Josh Kuhn Chris Gage
Department of Computer Science and Engineering
University of South Florida
Tampa, Florida 33620
{ligatti, jakuhn2, cgage}@cse.usf.edu

***Abstract*—We present 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 a 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.**

***Keywords*-computer networks; algorithms.**

I. 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 enable any incoming packet p to be mapped 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).

A. Related Work

Many algorithms exist for packet classification; several articles and books survey the field (e.g., [18], [14], [2], [9]). Existing algorithms typically only handle range or prefix patterns in class specifications (e.g., [8], [6], [13], [1], [11]). 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 may require 2^{b-1} range/prefix patterns to be written equivalently (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 [3].

On the other hand, range/prefix patterns can be converted (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) [17]. Bitmask patterns are therefore more efficiently expressive, in general, than range/prefix patterns alone.

Some classification solutions can handle noncontiguous-bitmask patterns, such as Recursive Flow Classification (RFC) [3]. Although RFC also handles range patterns, it requires prohibitively large amounts of time and memory when more than about 6000 rules exist [3]. Ternary content-addressable memories (TCAMs) can also classify packets according to noncontiguous-bitmask rules and are a popular technology for doing so. However, TCAMs are expensive, special-purpose hardware with limited capacity (up to about 4.5MB) for storing rules [7], [10].

Other classification algorithms handle patterns more expressive than bitmasks, including full regular expressions (e.g., EFSAs [15], XFAs [16], and BDDs [4]). However, all algorithms in this category suffer from worst-case exponential (in the number of packet bits being classified) memory requirements and/or classification times, due to the arbitrarily complex set of packets specifiable in a single rule.

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, the space used and packet-classification times are both $O(nb/w)$, as each of the n rules may specify b packet bits that have to be matched (assuming bitmask rules), with those 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.

B. Contributions

This paper presents an algorithm called Grouper (described in detail in Section II) 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. The set of rules matching any packet is computed 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 [6], [3], [1], 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 [5], good performance even with worst-case rule sets, and extendability to handle range rules and dynamic updates to rule sets.

As described in Section III, experiments have found Grouper, when implemented in software on a commodity laptop using about 2GB of memory, capable 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.

II. The Grouper Algorithm

Grouper uses t lookup tables to classify b packet bits according to n rules. Each lookup table maps either $\lfloor b/t \rfloor$ or $\lceil b/t \rceil$ 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.

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., 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 1 in that final bitmap indicates the lowest-numbered (highest-priority) rule matching the original b input bits.

Because Grouper does not guarantee that any two bits will be grouped 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 the set of rules a particular bit value matches cannot be influenced by other packet bits; it is this constraint that limits Grouper (in its basic version) to classifying according to bitmask rules.

Grouper does however guarantee that the b packet bits are partitioned 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 (whose bitmap entries would indicate which rules match a disproportionately small number of packet bits but would nonetheless have to be looked up and intersected just like those of the larger tables).

A. 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 a linear search is performed.

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).

B. 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 number of bits m required to store Grouper's tables is defined by the following equation.

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

Equation 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.

Classification time is $O(tn/w)$ because Grouper queries every one of the t tables to obtain a bitmap consuming $O(n/w)$ memory words, and as each of those bitmaps is fetched, it gets 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.

C. 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 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 1 with the given b and n values, produces a memory requirement no greater than the given maximum m value. This binary-search algorithm produces the optimal t in $O(\lg b)$ time.

III. Empirical Analysis

A. Implementation

We implemented a prototype of Grouper in 1093 lines of C code. The source code and benchmarking scripts are available online [5]. 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.

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.

B. 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 IPv4 addresses), 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 IPv6 header), and 12,000 (corresponding to classification based on the entire contents of a maximum-sized Ethernet payload). The n values tested were: 100, 1K, 10K, 100K, and 1M (in this paper, 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 independent 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. The throughput measurement was made by creating a file of 500K random b-length packets (but only 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 subsection, any throughputs reported in terms of bps have been calculated from the original pps measurement using a fixed packet size of 12K bits. Table-build times were also measured in real time, and all tests were performed three times (the results reported here are averages of the three trials).

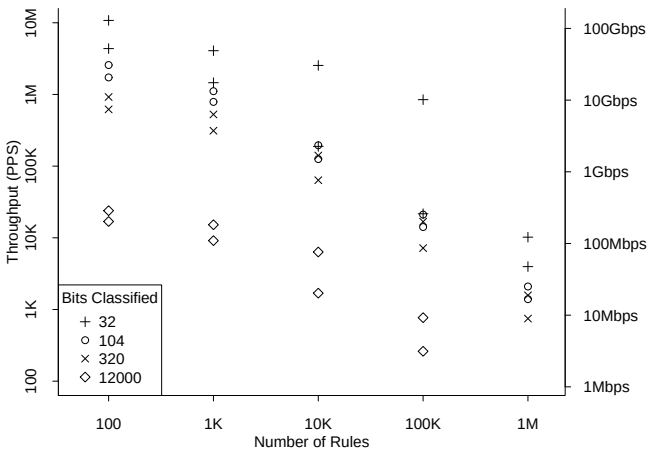

Figure 1. Maximum and minimum classifier throughputs.

C. Results

The graphs in Figures 1–2 summarize our experimental results. These figures present throughputs and table-build times for given values of n and b. 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 1 shows that our prototype software implementation performs well, particularly given that rule sets often have fewer than 1K rules [3], [19]. Figures 1–2 (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 3–5 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, even as the amount of memory used increases, are due to the laptop's 4MB L2 cache size.

Finally, Figures 6–7 depict Grouper's performance in a couple extreme cases of large numbers of bits being classified and/or large numbers of rules. These graphs illustrate the "y=1/x" relationship between throughput (y-axis) and number of tables (x-axis), which results from Grouper's $O(tn/w)$ classification time per packet.

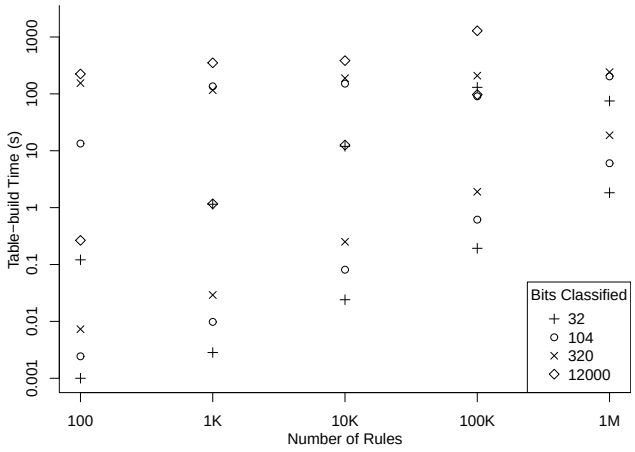

Figure 2. Maximum and minimum table-build times.

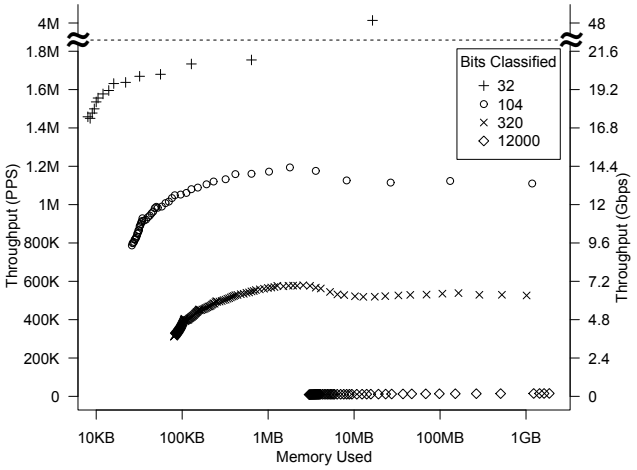

Figure 3. Throughputs for 1,000 rules.

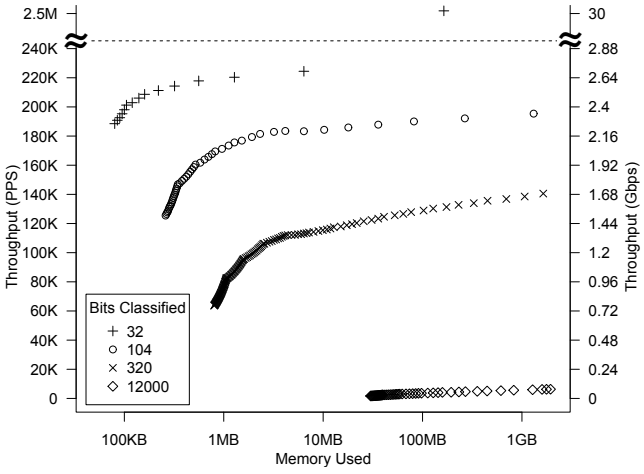

Figure 4. Throughputs for 10,000 rules.

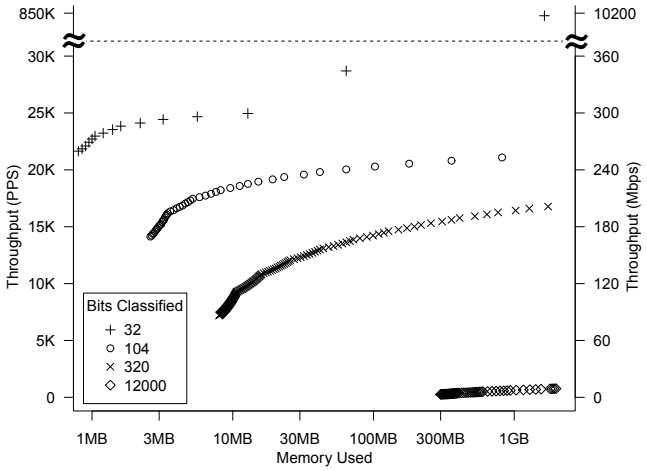

Figure 5. Throughputs for 100,000 rules.

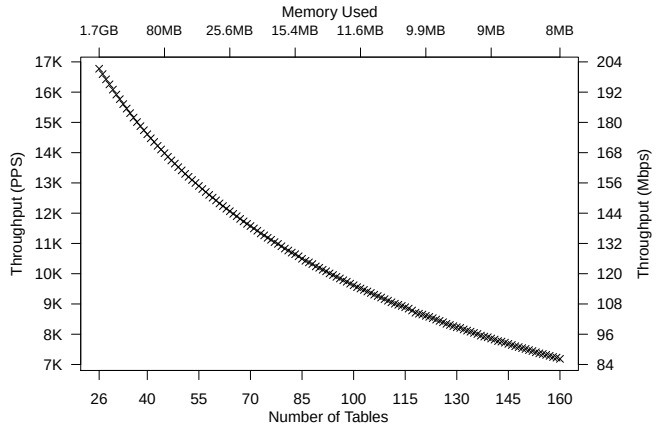

Figure 6. Throughputs for 320 bits classified, with 100,000 rules.

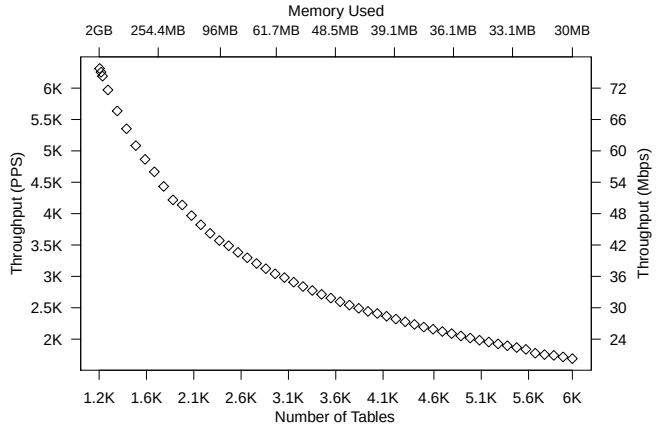

Figure 7. Throughputs for 12,000 bits classified, with 10,000 rules.

IV. Summary and Discussion of Extensions

We have 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. 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). The only characteristics of rule sets that affect Grouper's performance are the b and n values; the actual bitmasks used do not affect performance. Hence, our experimental results on randomly generated rule sets demonstrate Grouper's performance on worst-case rule sets.

We are considering two extensions to Grouper. The first would, in some cases, handle range rules by grouping together all the packet bits involved in each range. For example, Grouper could handle port-number ranges by ensuring that all bits corresponding to a port number are used to index the same table. For worst-case rule sets, this strategy is significantly more efficient than expanding ranges into bitmasks [12], as long as n is not too large relative to m (e.g., at most about 100K rules when 2GB of space is available).

The second extension would handle dynamic updates to rule sets, for those cases where rebuilding lookup tables from scratch takes too long (cf. Figure 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 are needed initially. If this extra space ever gets exhausted, Grouper could use another thread to rebuild the tables without taking the classification thread offline.

Acknowledgments

We're grateful for the helpful feedback from the anonymous reviewers of IEEE ICCCN 2010 and ACM SAC 2009. This research was supported by NSF grants CNS-0716343 and CNS-0742736.

References

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

[2] P. Gupta. Multi-dimensional packet classification. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, chapter 49. Chapman & Hall/CRC, 2005.

[3] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proceedings of SIGCOMM*, 1999.

[4] S. Hazelhurst, A. Attar, and R. 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.

[5] J. Kuhn, J. Ligatti, and C. Gage. The grouper webpage. http://www.cse.usf.edu/~ligatti/projects/grouper/.

[6] 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.

[7] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for advanced packet classification with ternary CAMs. In *Proceedings of SIGCOMM*, 2005.

[8] W. Lu and S. Sahni. Succinct representation of static packet classifiers. In *Proceedings of the IEEE Symposium on Computers and Communications*, 2007.

[9] D. Medhi and K. Ramasamy. *Network Routing: Algorithms, Protocols, and Architectures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[10] C. R. Meiners, A. X. Liu, and E. 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, Oct. 2009.

[11] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. Packet classification algorithms: From theory to practice. In *Proceedings of Infocom*, 2009.

[12] O. Rottenstreich and I. Keslassy. Worst-case TCAM rule expansion. In *Proceedings of Infocom*, 2010.

[13] 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.

[14] S. Sahni, K. S. Kim, and H. Lu. IP router tables. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, chapter 48. Chapman & Hall/CRC, 2005.

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

[16] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 187–201, 2008.

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

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

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