# The Geometric Efficient Matching Algorithm for Firewalls

Dmitry Rovniagin and Avishai Wool, *Senior Member, IEEE*

**Abstract**—Since firewalls need to filter all the traffic crossing the network perimeter, they should be able to sustain a very high throughput, or risk becoming a bottleneck. Firewall packet matching can be viewed as a point location problem: Each packet (point) has 5 fields (dimensions), which need to be checked against every firewall rule in order to find the first matching rule. Thus, algorithms from computational geometry can be applied. In this paper we consider a classical algorithm that we adapted to the firewall domain. We call the resulting algorithm "Geometric Efficient Matching" (GEM). The GEM algorithm enjoys a logarithmic matching time performance. However, the algorithm's theoretical worst-case space complexity is $O(n^4)$ for a rule-base with $n$ rules. Because of this perceived high space complexity, GEM-like algorithms were rejected as impractical by earlier works. Contrary to this conclusion, this paper shows that GEM is actually an excellent choice.

Based on statistics from real firewall rule-bases, we created a Perimeter rules model that generates random, but non-uniform, rule-bases. We evaluated GEM via extensive simulation using the Perimeter rules model. Our simulations show that on such rule-bases, GEM uses near linear space, and only needs approximately 13MB of space for rule-bases of 5,000 rules. Moreover, with use of additional space improving heuristics, we have been able to reduce the space requirement to 2-3MB for 5,000 rules.

But most importantly, we integrated GEM into the code of the Linux `iptables` open-source firewall, and tested it on real traffic loads. Our `GEM-iptables` implementation managed to filter over 30,000 packets-per-second on a standard PC, even with 10,000 rules. Therefore, we believe that GEM is an efficient, and practical, algorithm for firewall packet matching.

**Index Terms**—Network Communication, Network-level security and protection

## 1 Introduction

### 1.1 Motivation

The firewall is one of the central technologies allowing high-level access control to organization networks. Packet matching in firewalls involves matching on many fields from the TCP and IP packet header. At least five fields (protocol number, source and destination IP addresses, and ports) are involved in the decision which rule applies to a given packet. With available bandwidth increasing rapidly, very efficient matching algorithms need to be deployed in modern firewalls to ensure that the firewall does not become a bottleneck.

Modern firewalls all use "first match" semantics [5], [40], [43]: The firewall rules are numbered from 1 to $n$, and the firewall applies the policy (e.g., pass or drop) associated with the first rule that matches a given packet. See Fig 1 for an illustrated example.

Firewall packet matching is reminiscent of the well studied router packet matching problem. However, there are several crucial differences which make the problems quite different. First, unlike firewalls, routers use "longest prefix match" semantics. Next, the firewall matching problem is 4- or 5-dimensional, whereas router matching is usually 1- or 2-dimensional: A router typically matches only on IP addresses, and does not look deeper, into the TCP or UDP packet headers. Finally, major firewall vendors support rules that utilize IP address *ranges*, in addition to subnets or CIDR blocks:[1] this is the case for Check Point and Juniper—the main exception is Cisco, that only supports individual IP addresses or subnets. Therefore, firewalls require their own special algorithms.

### 1.2 Statefull Firewall Matching

Most modern firewalls are stateful. This means that after the first packet in a network flow is allowed to cross the firewall, all subsequent packets belonging to that flow, and especially the return traffic, is also allowed through the firewall. This statefulness has two advantages. First, the administrator does not need to write explicit rules for return traffic—and such return-traffic rules are inherently insecure since they rely on source-port filtering (see discussion in [43] and Check Point's patent [29]). So stateful firewalls are fundamentally more secure than simpler, stateless, packet filters. Second, state lookup algorithms are typically simpler and faster than rule-match algorithms, so statefulness potentially offers important performance advantages.

Firewall statefulness is commonly implemented by two separate search mechanisms: (i) a slow algorithm that implements the "first match" semantics and compares a packet to all the rules, and (ii) a fast state lookup mechanism that checks whether a packet belongs to an existing open flow. In many firewalls, the slow algorithm is a naive linear search of the rule-base, while the state lookup mechanism uses a

*Part of this work has appeared, in extended abstract form, in the 23rd convention of IEEE Israel.*

*D. Rovniagin and A. Wool are with the School of Electrical Engineering, Tel Aviv University, Ramat Aviv 69978, Israel. E-mail: dmitry.rovniagin@gmail.com, yash@acm.org*

1. It is possible to convert an arbitrary range of IP addresses into a collection of subnets—however, as many as 62 subnets may be necessary to cover a single IP address range, thus there is a great loss of efficiency in the conversion.

```
access-list 101 permit tcp 12.20.51.0 255.255.255.0 host 1.2.3.4 gt 0
access-list 101 deny tcp 12.20.51.0 255.255.255.0 1.2.0.0 255.255.0.0 eq 135
```

Fig. 1. Excerpts from a Cisco PIX firewall configuration file, showing 2 rules. Both rules refer to the TCP protocol. The source in both rules is the same subnet. The first rule has a single IP address as a destination but a range of destination ports (1–65535), while the second rule has a range of destination IP addresses but a single destination port. Note that a TCP packet with source IP 12.20.51.1, destination IP 1.2.3.4, and destination port 135 matches both rules, but because of the first-match semantics, the first rule's decision ("permit") is triggered.

hash-table or a search-tree: This is the case for the open-source firewalls `pf` [24] and `iptables` [23]. There are strong indications that commercial firewalls use linear search for the slow rule-match as well: E.g., Check Point rules are translated into an assembly-like macro language called INSPECT [40] with linear semantics, and the INSPECT language is simply translated into bytecode. Moreover, the standard advise for improving firewall performance, for all vendors, is to place the most popular rules near the top of the rule-base (cf. [14], [7]). This advise doesn't make much sense if the firewall rearranges the rules into a complex search data structure.

Note that a stateful firewall's two-part design provides its highest performance on long TCP connections, for which the fast state lookup mechanism handles most of the packets. However, connectionless UDP[2] and ICMP traffic, and short TCP flows, like those produced in extremely high volume by Distributed Denial of Service attacks (cf. [19]), only activate the "slow" algorithm, making it a significant bottleneck. Our main result is that the "slow" algorithm does *not* need to be slow, even in a software-only implementation running on a general-purpose operating system. We show that the GEM algorithm has a matching speed that is comparable to that of the state lookups: In isolation the algorithm required under $1\mu$sec per packet, and our Linux `GEM-iptables` implementation sustained a matching rate of over 30,000 packets-per-second (pps), with 10,000 rules, without losing packets, on a standard PC workstation.

### 1.3 Contributions

In this paper we revisit a classical algorithm from computational geometry (cf. [10], [22]), and apply it to the firewall packet matching. In the firewall context we call this algorithm the Geometric Efficient Matching (GEM) algorithm. This algorithm performs matching in $O(d \log n)$ time, where $n$ is the number of rules in the firewall rule-base and $d$ is the number of fields to match. The worst-case space complexity of GEM is $O(n^d)$. For instance, for TCP and UDP we have $d = 4$, giving a search time of $O(\log n)$ and worst case space complexity of $O(n^4)$.

The GEM data structure allows easy control over the order of fields to be matched. The data structure can be used for any number of dimensions $d$, but typical values for firewall packet matching are either $d = 2$ for opaque protocols like IPsec (protocol 50 or 51) or $d = 4$ for TCP, UDP, and ICMP. We focus on the more difficult case for the algorithm, with $d = 4$, in which the match fields are: source IP address, destination IP address, and source and destination port numbers. This fits TCP and UDP filtering, and also ICMP (using the 8-bit message type and code instead of 16-bit port numbers).

Note that the worst-case space complexity can only be caused by an unlucky rule-base structure, and not by the packets that the firewall encounters. Furthermore, knowledge of the rule-base does not help an attacker force the firewall into poor performance since the search time is deterministically logarithmic in the worst case—so GEM is not subject to algorithmic complexity attacks [8], [3].

To address the worst-case space complexity, we propose two approaches. One approach involves optimization heuristics. The other is a time-space trade-off, which at the cost of a factor $\ell$ slowdown in the search time, provides an $\ell^{d-1}$ decrease in the space complexity.

The next step in our evaluation of the GEM algorithm was an extensive simulation study. Our simulations showed that, in isolation, the algorithm required under $1\mu$sec per packet, on a standard PC, even for rule-bases of 10,000 rules. Furthermore, we found that the worst case space complexity manifests itself when the rule-base consists of uniformly-random rules.

However, real firewall rule-bases are far from random. Rule-bases collected by the Lumeta (now AlgoSec) Firewall Analyzer [42], [44] show that, e.g., the source port field is rarely specified, and the destination port field is usually a single port number (not a range) taken from a set of some 200 common values.

Based on statistics we gathered from real rule-bases, we created a non-uniform model for random rule-base generation, which we call the Perimeter rule model. On rule-bases generated by this model, we found that the order of field evaluation has a strong impact on the data structure size (several orders of magnitude difference between best and worst). We found that the evaluation order which results in the minimal space complexity is: destination port, source port, destination IP address, source IP address. With this evaluation order, the growth rate of the data structure is nearly linear with the number of rules. The data structure size for rule bases of 5,000 rules is $\approx 13MB$, which is entirely practical. Using more aggressive space optimizations allows us to greatly reduce the data structure at a cost of a factor of 2 or 3 slowdown. For instance, using 3-part heuristic division, we get a data structure size of 2MB for 10,000 rules.

Beyond simulations, we created a fully functional GEM implementation within the Linux `iptables` open-source

2. Some firewalls treat UDP traffic as connection-oriented and perform state lookups on UDP packets as well.

TABLE 1
Header field numbering.

| number | description | space |
|---|---|---|
| 0 | source IP address | 32bit |
| 1 | destination IP address | 32bit |
| 2 | source port number | 16bit |
| 3 | destination port number | 16bit |
| 4 | protocol | 8bit |

firewall [23], and tested its performance in a laboratory testbed. Our `GEM-iptables` Linux implementation sustained a matching rate of over 30,000 pps, with 10,000 rules, without losing packets. In comparison, the non-optimized `iptables` could only sustain a rate of $\approx 2500$ pps with the same rule-base.

Thus, we conclude that the GEM algorithm is an excellent, practical, algorithm for firewall packet matching: Its matching speed is far better than the naive linear search, and its space complexity is well within the capabilities of modern hardware even for very large rule-bases.

Parts of this work have appeared, in extended abstract form, in [28].

**Organization:** Section 2 formally defines the matching problem and describes the GEM algorithm along with its data structure. Section 3 describes the statistics we gathered from firewall rule-bases. Section 4 introduces the non-uniform Perimeter rule model, and describes the simulation results in this model. Section 5 describes our `iptables` implementation and the performance it achieved. Section 6 describes the time-space trade-off and space optimizations. Section 7 describes related work, and we conclude with Section 8.

## 2 The Algorithm

### 2.1 Definitions

The firewall packet matching problem finds the first rule that matches a given packet on one or more fields from its header. Every rule consists of set of ranges $[l_i, r_i]$ for $i = 1, \ldots, d$, where each range corresponds to the $i$-th field in a packet header. The field values are in $0 \leq l_i, r_i \leq U_i$, where $U_i = 2^{32} - 1$ for IP addresses, $U_i = 65535$ for port numbers, and $U_i = 255$ for ICMP message type or code. Table 1 lists the header fields we use (the port fields can double as the message type and code for ICMP packets). For notation convenience later on, we assign each of these fields a number, which is also listed in the table.

**Remarks:**

- Most firewalls allow matching on additional header fields, such as IP options, TCP flags, or even the packet payload (so called "deep packet inspection"). However, real rule-bases [44] very rarely use such futures. Nearly all the firewall rules that we have seen only refer to the five fields listed in Table 1.
- The description above, and the GEM algorithm, is mostly suitable to firewalls whose rules use *contiguous* ranges of IP addresses. This is not a limitation for enterprise firewalls—we have never encountered an enterprise firewall that uses non-contiguous masks.
- We use '$*$' to denote wildcard: An '$*$' in field $i$ means any value in $[0, U_i]$.
- We are ignoring the action part of the rule (e.g., pass or drop), since we are only interested in the matching algorithm.

### 2.2 The Sub-Division of Space

In one dimension, each rule defines one range, which divides space into at most 3 parts. It is easy to see that $n$ possibly overlapping rules define a subdivision of one-dimensional space into at most $(2n - 1)$ *simple ranges*. To each simple range we can assign the number of the *winner* rule. This is the first rule which covers the simple range.

In $d$-dimensions, we pick one of the axes and project all the rules onto that axis, which gives us a reduction to the previous one-dimension case, with a subdivision of the one dimension into at most $(2n - 1)$ simple ranges. The difference is that each simple range corresponds to a set of rules in $(d - 1)$ dimensions, called *active rules*. We continue to subdivide the $(d-1)$ dimensional space recursively. We call each projection onto a new axis a *level* of the algorithm, thus for a $4$-dimensional space algorithm we have $4$ levels of subdivisions. The last level is exactly a one-dimensional case—among all the active rules, only the winner rule matters.

At this point we have a subdivision of $d$-dimensional space into simple hyper-rectangles, each corresponding to single winning rule. In Section 2.4 we shall see how to efficiently create this subdivision of $d$-dimensional space, and how it translates into an efficient search structure.

### 2.3 Dealing with the Protocol Field

Before delving into the details of the search data structure, we first consider the protocol header field. The protocol field is different from the other four fields: very few of the 256 possible values are in use, and it makes little sense to define a numerical "range" of protocol values. This intuition is validated by the data gathered from real firewalls (see Section 3): The only values we saw in the protocol field in actual firewall rules were those of specific protocols, plus the wildcard '$*$', but never a non-trivial range.

Thus, the GEM algorithm only deals with single values in the protocol field, with special treatment for rules with '$*$' as a protocol. We preprocess the firewall rules into categories, by protocol, and build a separate search data structure for each protocol (including a data structure for the '$*$' protocol). The actual geometric search algorithm only deals with $4$ fields.

Now, a packet can only belong to one protocol—but it is also affected by protocol = '$*$' rules. Thus every packet needs to be searched twice: once in its own protocol's data structure, and once in the '$*$' structure. Each search yields a candidate winner rule.[3] We take the action determined by the candidate with the lower number.

In the remainder of this paper, we focus on the TCP protocol, which has $d = 4$ dimensions, although the same

3. If no rule matches, we assume that the packet matches an implicit default catch-all rule with a maximal rule-number.

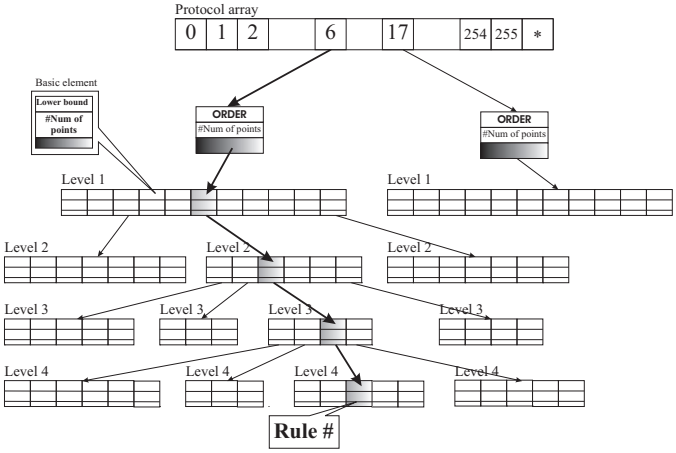


Fig. 2. Overall GEM data structure overview.

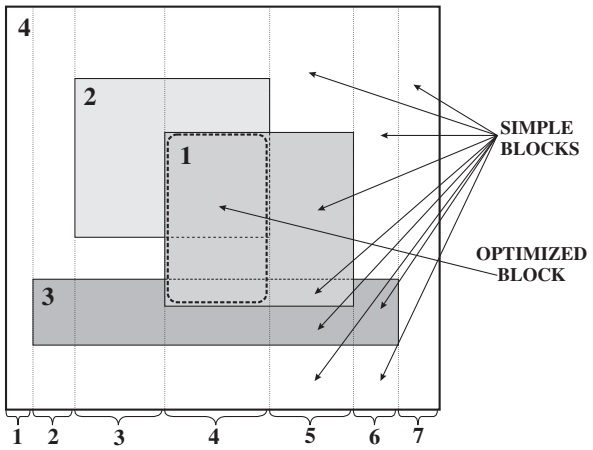


Fig. 3. The last two levels of building the search data structure. At this point the rules are two-dimensional, e.g., the X axis may represent the destination IP and the Y axis is the destination port. We can see three rules, shown as shaded overlapping rectangles, plus the default rule in white. The critical points and simple ranges are projected onto the X axis. Three blocks in rule 1 are optimized.

discussion applies for UDP and ICMP. In Section 3 we shall see that TCP alone accounts for 75% of rules on real firewalls, and collectively, TCP, UDP, and ICMP account for 93% of the rules.

### 2.4 The Data Structure

The GEM search data structure consists of three parts. The first part is an array of pointers, one for each protocol number, along with a cell for the '$*$' protocol (as mentioned in Section 2.3). We build the second and third parts of the search data structure for each protocol separately.

The second part is a *protocol database header*, which contains information about the *order* of data structure levels. The order in which the fields of packet header are checked is encoded as a $4$-tuple of field numbers, using the numbering of Table 1. The protocol database header also contains the pointer to the first level and the number of simple ranges in that level.

The third part represents the levels of data structure themselves. Every level is a set of nodes, where each node is an array. Each array cell specifies a simple range, and contains a pointer to the next level node. In the last level the simple range information contains the number of the winner rule instead of the pointer to the next level. See Fig 2 for an illustration.

The basic cell in our data structure (i.e., an entry in the sorted array which is a node in the structure) has a size of 12 bytes: 4 for the value of the left boundary of the range, 4 for the pointer to the next level, and 4 for the number of cells in the next-level node. The nodes at the deepest level are slightly different, consisting of only 8 bytes: 4 for the left boundary of the range and 4 for the number of winner rule.

Note that the order of levels is encoded in the protocol database header, which gives us convenient control over the field evaluation order.

### 2.5 The Search Algorithm

The packet header contains the protocol number, source and destination address and port numbers fields. First, we check the protocol field and go to the protocol array of the search data structure, to select the corresponding protocol database header. From this point, we apply a binary search with the corresponding field value on every level, in order to find the matching simple range and continue to the next level. The last level will supply us with the desired result—the matching rule number.

For example, suppose we have an incoming TCP packet. Assume that the GEM protocol header for TCP shows that the order of levels is 1203. The first level - 1 - denotes the destination address. We execute a binary search of the destination address value from packet header against the values of the array in the first level. The simple range associated with the found array item points us to the corresponding node from the second level. The second level, in our example (2) denotes the source port number. By binary search on the second level array we find a new simple range, which contains the packet source port number. Similarly, we search for the source address (field 0) and destination port (field 3). In the last level node we find the winner rule information.

We repeat the search procedure for protocol '$*$', and get another "winner" rule. From the two candidates we choose the one with the lower rule number.

**Search time:** In each level we execute a binary search on an array of at most $2n$ entries, where $n$ is the maximal number of active rules. We process two searches: one with the packet's protocol and one in the '$*$' data structure. Thus, for $d$ levels, the search time is $O(d \log n)$. For a constant $d = 4$, we get an $O(\log n)$ search time. Note that the '$*$' search data structure only has 2 levels (for IP addresses), thus the search time is dominated by the time to search the 4 levels of the TCP search data structure.

TABLE 2
Protocol and port numbers distribution in rule-bases.

| SOURCE PORT DISTRIBUTION | |
|---|---|
| * | 98% |
| ranges | 1% |
| single port | 1% |

| PROTOCOL DISTRIBUTION | |
|---|---|
| * | 6% |
| TCP | 75% |
| UDP | 14% |
| ICMP | 4% |
| Other | 1% |

| DESTINATION PORT DISTRIBUTION | | |
|---|---|---|
| * | 0% | |
| ranges | 4% | |
| average range size | 27030 | |
| single ports | 96% | |
| average # single ports per rule-base | 50 | |
| most referred-to ports in rules | 80 | 6.89% |
| | 21 | 5.65% |
| | 23 | 4.87% |
| | 443 | 3.90% |
| | 8080 | 2.25% |
| | 139 | 2.16% |

### 2.6 The Build Algorithm

The build algorithm is executed once for each protocol. The input to the build algorithm consists of the rule-base, plus the field order to use. The order dictates the contents of each data structure level, and also, the order in which the header fields will be tested by the search algorithm. There are $4! = 24$ possible orders we can choose from, to check 4 fields. The data structure is built using a geometric sweep-line algorithm (cf. [9]).

All four levels of the search data structure are built in the same manner. We start with the set of *active* rules from the previous level. For the first level all the rules with the specified protocol (e.g., TCP) are active.

We then construct the set of *critical points* of this level—these are the endpoints of the ranges, which are the projections of the active rules onto the axis that corresponds to the currently checked field (See Fig 3). For example, if the first field is "1" (destination IP address), then the critical points are all the IP addresses that start or end a destination IP address range in any rule. We sort the list of critical points in increasing order, and run the sweep-line over them. Note that there are two implicit critical points: 0, and the maximal value for the level. Every critical point corresponds to a start of one simple range, which in turn relates to a subset of active rules.

For each simple range we calculate its set of active rules, by choosing all the rules that overlap the simple range in the current field. For example, in Fig 3, rules 2, 3 and 4 are relevant for the third simple range on the X axis. With this new set of active rules we continue to the next level for each one of the simple ranges. In the deepest level we only need to list the number of the "winner rule": the rule with lowest number among the active rules associated with the current range.

**Build time and space complexity:** In the worst case, GEM performs a sort of $\Omega(n)$ values for each of the $d$ levels, giving a build time complexity of $O((n \log n)^d)$. It is easy to see that the space complexity is $O(n^d)$ in the worst case, and $O(n^4)$ for TCP or UDP.

### 2.7 Reducing The Space Usage: Basic Optimizations

A space complexity of $O(n^4)$ may be theoretically acceptable since it is polynomial. However, with $n$ reaching thousands of rules [44], conserving space is crucial. Here we introduce two optimization heuristics, which significantly reduce GEM's space requirement.

The first optimization works on the last level of the data structure. If we take a closer look at last level ranges, we see that occasionally two or more adjacent ranges point to the same "winner" rule. This means that we can replace all these ranges with a single range which is their geometrical union (see Fig 3).

The second optimization works on the one-before-last level of the search data structure. Occasionally, there exist simple ranges that point to equivalent last level structures. Instead of storing the same last level structure multiple times, we keep a single last level structure, and replace the duplicates by pointers to the main copy. For example, in Fig 3, ranges 2 and 6 are equivalent (rules 4-3-4, with boundaries in the same vertical positions)

As part of the simulation study, we tested the effectiveness of these optimizations. Our simulations on rule bases of sizes from 500 to 10,000 show that the optimizations reduce the search data structure size by 30%$\sim$60% on average, and that the effect grows with rule-base size (See Section 4.4.2).

We also tried to apply this optimization method on the higher levels of our data structure, but we found that this greatly increases the preprocessing time, and only gives minor improvements to the space complexity. We omit the details. Some space/time optimization tradeoffs are discussed in section 6. We remark that additional optimization techniques for GEM-like data structures are known to perform well in the computational geometry literature, so it would be interesting to test their effectiveness in the firewall matching domain. Possibilities include: not using the same field ordering in every branch of the search tree; switching to the next branch before completing the search along an axis; or even replacing the last two levels of binary search tree with a data structure optimized for two-dimensional queries such as that of [11] or [4].

## 3 Firewall Rule-Base Statistics

To get a better understanding of what real-life firewall rule-bases look like, we gathered statistics from firewall rule-bases that were analyzed by the Lumeta (now AlgoSec) Firewall Analyzer [42], [44]. The statistics are based on 19 rule-bases from enterprise firewalls (Cisco PIX and Check Point FireWall-1)

collected during 2001 and 2002. The rule-bases came from a variety of corporations from the financial, telecommunications, automotive, and pharmaceutical industries. We analyzed a total of 8434 rules.

Table 2 shows the distribution of protocols in the rules we analyzed. The data shows that $75\%$ of rules from typical firewall rule-bases match TCP, and a total of 93% match TCP, UDP or ICMP. Of these the most important is clearly TCP. Therefore, we concentrate on these protocols in the rest of paper. In our problem context, these protocols are the most difficult for evaluation since they imply a $4$-dimensional space.

The same table shows the distribution of TCP source and destination port numbers. We can clearly see that the source port number is rarely specified: 98% of the rules have a wildcard '*' in the source port. This makes sense because both PIX and FireWall-1 are stateful firewalls that do not need to perform source-port filtering to allow return traffic through the firewall—and source port data is generally unreliable because it is usually under the control of the attacker.

On the other hand, the TCP destination port is usually specified precisely. The vast majority of rules specified a single port number, but 4% allowed a range of ports, and the ranges tended to be quite large. Common ranges are "all high ports" (1024–65535) and "X11 ports" (6000-6003). The single port numbers we encountered were distributed among some 200 numbers, the most popular of which are shown in Table 2: these correspond to the HTTP, FTP, Telnet, HTTPS, HTTP-Proxy, and NetBIOS services.

# 4 The Simulation Study

## 4.1 The Random Rules Simulation

As the first step of our performance evaluation of GEM we implemented and tested it in isolation. The GEM build and search algorithms were implemented in C using Microsoft VC++ 6.0. The simulations were performed on a 733MHz Pentium III PC with 256MB of RAM running the Windows XP operating system.

We started by testing GEM using uniformly-generated rules: for every rule, each endpoint of each of the 4 fields (IP address ranges and port ranges) was selected uniformly at random from its domain. We built the GEM data structure for increasing numbers of such rules and then used the resulting structure to match randomly generated packets. We omit the details for lack of space, and instead refer the reader to [27].

On one hand, these early simulations showed us that the search itself was indeed very fast: a single packet match took around $1\mu$sec, since it only required 4 executions of a binary search in memory.

On the other hand, we learned that the data structure size grew rapidly—and that the order of fields had little or no effect on this size. The problem was that since the ranges in the rules were chosen uniformly, almost every pair of ranges (in every dimension) had a non-empty intersection. All these intersections produced a very fragmented space subdivision, and effectively exhibited the worst-case behavior in the data structure size. We concluded that a more realistic rule model is needed.

## 4.2 The Perimeter Rules Model

As we saw in Section 3, real firewall rule-bases have a large degree of structure. Thus, we hypothesized that realistic rule-bases rarely cause worst-case behavior for the GEM algorithm. Furthermore, we wanted to test the effects of the field order on the performance of GEM on such rule-bases. For this purpose, we built the Perimeter firewall rules model, and simulated the behavior of GEM on rule-bases generated in this model.

### 4.2.1 The Modeled Topology

The model assumes a perimeter firewall with two "sides": a protected network on the inside, and the Internet on the outside. The inside network consists of 10 class B networks, and the Internet consists of all other IP addresses. Thus, the internal network contains $10 \cdot 65536$ possible IP addresses. In reality, organizations that actually own 10 class B networks are quite rare. However, we used this assumption for two reasons:

1) Many organizations use private (RFC 1918) IP addresses internally, and export them via network address translation (NAT) on outbound traffic. Such organizations often use large subnets liberally, e.g., assign a 172.x.*.* class B subnet to each department.
2) Having a large internal subnet stresses the GEM algorithm since we pick random ranges from the internal ranges.

### 4.2.2 The Rules

The Perimeter rules model produces rules of two types: Inbound rules, that allow traffic from the Internet into the protected network, and Outbound rules, that allow traffic from the protected network out to the Internet. Each rule in the rule-base is constructed randomly according to the distribution detailed in Table 3 for its type (Inbound or Outbound).

**Inbound rules**. When we are modeling rules for inbound traffic, the source IP addresses are rarely specified in the rules, and 95% of the rules have '$*$' as their source address. The remaining 5% have a range in their source address field, chosen uniformly at random from the Internet's IP addresses. The destination addresses for inbound rules are always internal, belonging to the 10 internal class B subnets. 45% of the rules have a randomly chosen individual internal IP address as a destination, modeling server machines. Another 15% have a *small random range*: a range which completely lies inside one of the internal class C networks. These ranges model clusters of servers and small classless subnets such as '/27's and '/28's. Then, 30% of the rules have a complete class C as a destination (i.e., a range of the form $a.b.c.0 - a.b.c.255$). Finally, 10% allow access to a full class B.

Note that Inbound rules produce many "collisions" in the destination field. E.g., consider the 30% of rules with a full class C destination. The Birthday paradox [13] shows that the probability of finding *some* class C destinations that collide is close to 1 when the number of rules exceeds $\sqrt{2560} \sim 50$. Essentially the same is true for collisions of a single-internal-IP-destination and an internal class C subnet, since every internal IP address has exactly a 1:2560 chance of falling inside a particular internal class C.

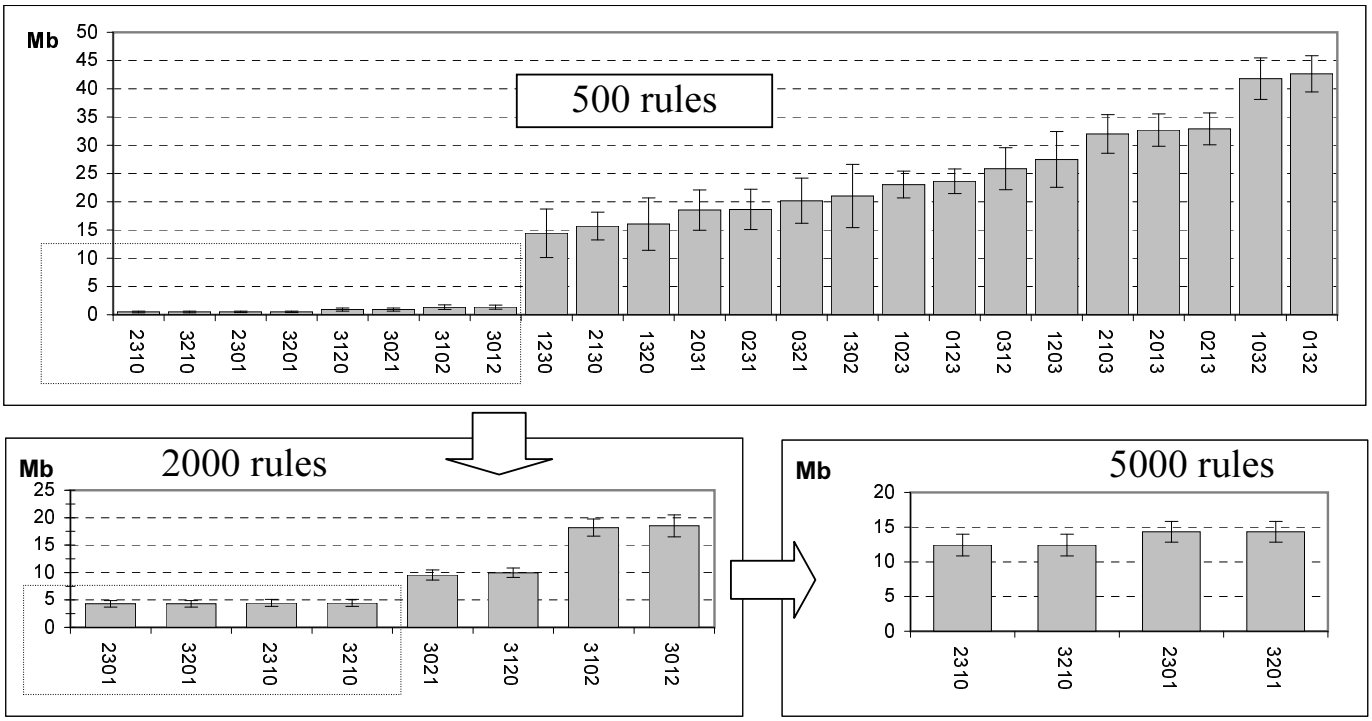


Fig. 4. Finding the best field order: GEM data structure size as a function of field order. The bars also show the 90% confidence intervals.

**Outbound rules**. When we are modeling the outbound rules, 90% of the rules have a destination IP address of '$*$'. 10% of the rules have either a specific address or a range in the destination field, modeling a rule that restricts or allows access to some particular server or network. The source addresses for outbound rules are selected from the internal addresses with the frequencies shown in Table 3.

**Services**. The service field in the rules is selected similarly for both Inbound and Outbound rules. The service is selected uniformly at random from a collection of 100 services, whose definitions were taken from real firewall rule-bases (recall Table 2). Most of these services have individual destination port numbers, however a few include port ranges, and one service is the '$*$' service. We allow a small rate of growth in the number of services by adding 2% of randomly generated services, where the destination port is randomly picked from 0 to 65535.

One concern we had was that, occasionally, the model generated a rule of the form "from $*$, to $*$, with service $*$".[4] When such a rule shows up in the rule-base, it acts as the default rule, and all subsequent rules become redundant, because of the "first match" semantics. This effectively shortens the rule-base, and prevents us from simulating GEM's behavior on large rule-bases. Thus, our model checks for, and discards, such randomly-generated catch-all rules.

The rule-bases generated by the model are still much less structured than actual firewall rule-bases. In real firewall rule-bases the number of internal servers is usually rather small, and they have many rules that refer to them. Also, it is considered insecure to allow many TCP services into large parts of the internal networks [44]. Both considerations would cause more repetitions in IP addresses, and hence, reduce the number of simple ranges, which would lead to smaller search data structures. Therefore, we believe our Perimeter model stresses the GEM algorithm more than real firewall rule-bases would.

4. This occurs with a probability of approximately 0.00025, so we can expect such a rule once every 4000 rules.

TABLE 3
The statistical distribution for rules in the Perimeter model. An '$*$' in the source IP address for Outbound rules represents all IP addresses inside the internal network.

| | | Inbound | Outbound |
|---|---|---|---|
| source IP | $*$ | 95% | 5% |
| | range | 5% | 15% |
| | Class B | | 10% |
| | Class C | | 25% |
| | single IP | | 45% |
| dest IP | $*$ | | 90% |
| | range | 15% | 5% |
| | Class B | 10% | |
| | Class C | 30% | |
| | single IP | 45% | 5% |
| service | from 100 services list | 96% | 96% |
| | dst port is random range | 2% | 2% |
| | dst port is single port | 2% | 2% |

### 4.3 Selecting the Best Field Order

Our first goal in the Perimeter model is to determine if any efficiency can be achieved by selecting the GEM data structure field order.

Preliminary simulations showed us that the order of fields had a very strong impact on the size of the data structure in

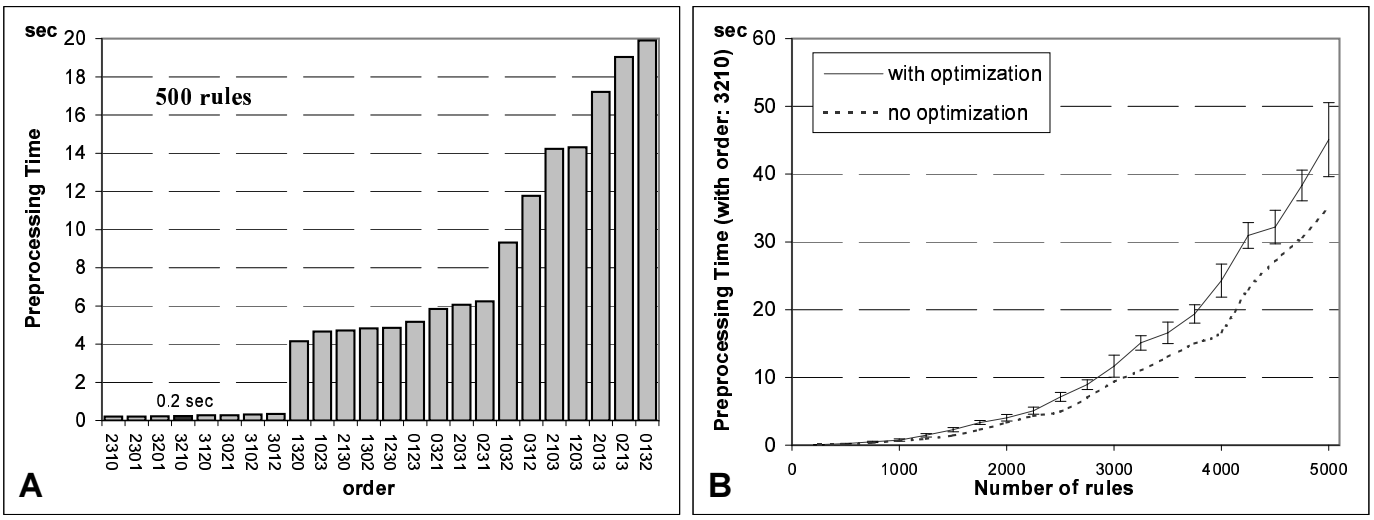


Fig. 5. (A): Build time as a function of the order of fields. (B): Build time as a function of the number of rules, with and without optimization, using the best (3210) order.

the Perimeter model (several orders of magnitude between best and worst choices). The variance was so large that we were unable to simulate the worst choices on large rule-bases, since the data structure grew to hundreds of MB and took up to 20 minutes to build.

The rationale is that the usage patterns in the different fields are non-uniform (as we saw in Section 3), so some choices of fields in the high levels of the hierarchy cause large amount of subdivisions in the lower levels (many ranges are created).

Therefore, we used a 3-stage process to identify the best order. In the first stage we generated small (500 rules) rule-bases, and built the data structure for each of the $4! = 24$ possible orders. This simulation showed that 16 orders were clearly much worse than others, so we dropped them and continued to 2000-rule sets with the remaining 8 orders. Here we found that the best 4 orders were better than the rest. The top 4 candidates were evaluated on 5,000-rule sets, which identified the best and second-best orders. The process of finding the best order for the "Perimeter" model is shown in Fig 4.

Fig 4 shows that the confidence intervals for the best 4 orders all overlap, indicating that the differences between them are not statistically significant. Moreover, a closer look shows that the position of field "2" (source port) among the best 8 orders is less significant: there are really only two orders (310 and 301) with the "2" field inserted in all 4 possible positions. This is reasonable because the source port in the Perimeter model is almost always '$*$', so it's position in the order has a limited impact. Therefore, for all subsequent tests we somewhat arbitrarily used the "natural" order of 3210 (destination port, source port, destination IP address, source IP address).

### 4.4 The Search Data Structure

Every point on the simulation result graphs represents the mean value from 10 independent runs. The graphs also show 90% confidence intervals (cf. [18]).

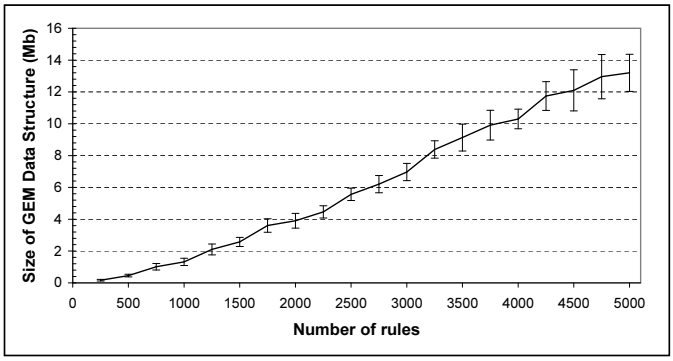


Fig. 6. Data structure size as a function of the number of rules.

#### 4.4.1 Growth Rate

After we identified the best field order, we investigated relatively large rule-bases to get a more precise picture of the GEM data structure and search algorithm properties. Fig 6 shows the GEM data structure size as a function of the rule-base size. As we can see, the data structure size grows almost linearly with the rule-base size, i.e., at a much slower rate than the theoretical upper bound of $O(n^4)$ indicates. By plotting the data on a $\log$-$\log$ scale and calculating a linear regression we found that the growth rate is $O(n^{0.95})$.

#### 4.4.2 Build Time

In this test we evaluated the time it takes to build the search data structure. Fig 5(A) shows the build time for different field orders. We compared all 24 orders on small-sized rule-bases (500 rules). Again we can see the great variability, with the fastest build about 2 orders of magnitude faster than the slowest. Luckily, our best field order also has a good build time ($4^{th}$ place).

Fig 5(B) shows the rate of growth in the build time. The figure shows that the build time grows at a super-linear rate, but that the time remains reasonable even for large rule-bases:

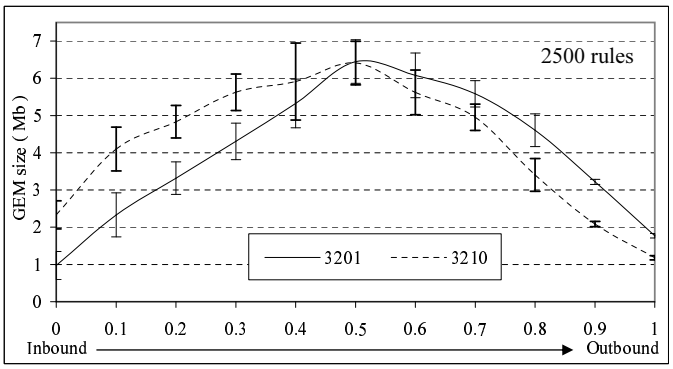


Fig. 7. The Inbound-Outbound Ratio vs. GEM data structure size. A ratio of 0 means that all rules are Inbound.

the search data structure for 5,000 rules took about 45 seconds to build. A linear regression of the $\log - \log$ plot shows a growth rate of $O(n^{1.7})$ with optimization and $O(n^{1.5})$ without optimization.

The figure also shows that about 20-30% of the build time is taken by the optimizations (recall Section 2.7). However, the optimizations give us 30-60% improvement in space usage of the GEM search structure. For example, if we use the best order and build the GEM data structure for 5,000 rules without optimization, it takes $\approx 20$ MB, rather than $\approx 13$ MB.

### 4.5 The Inbound - Outbound Ratio

An additional parameter of our Perimeter model is the ratio between the number of Inbound and Outbound rules. In order to determine the effect of this parameter, we ran the GEM building algorithm on rule-bases with different ratios of Inbound and Outbound rules. The results are shown in Fig 7. We show the results for two different field orders, that were among the best in Section 4.3.

The figure clearly shows that if the rules are homogeneous (ratios close to 0 or to 1), we get better space performance. The difference between homogeneous and mixed rule-bases can be up to a factor of 6 in size. In all subsequent tests we used an inbound-outbound ratio of 50% - again, to stress the GEM algorithm.

## 5 THE `GEM-iptables` IMPLEMENTATION

To evaluate GEM in a more realistic environment, we implemented the GEM algorithm and integrated it with the code of the Linux `iptables` firewall. We used Red Hat Linux 9 (kernel version 2.4.18-8) and `iptables` v1.2.8. We incorporated the GEM build algorithm into the user-space program `iptables`, and the GEM search algorithm into the `ip_tables` kernel module. The built GEM database was transferred from user space to the kernel using the mechanism already employed by `iptables`. We left the existing `iptables` linear search algorithm intact. The selection of linear or GEM search was controlled by a command line switch.

Since we wanted to be able to compare GEM's performance to the regular `iptables`, we adopted the `iptables` configuration language as our input. However, `iptables` does not support general ranges of IP addresses in the rules, and only accepts subnets. Therefore, we modified our rule generation module to only produce subnets, e.g., instead of generating a random IP range, we generate a random IP address and a random netmask that leaves the resulting subnet inside one class C network (recall Section 4.2). The modified rule generator output an `iptables` configuration script.

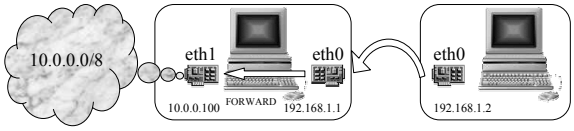


Fig. 8. Testbed system configuration.

### 5.1 Testbed setup

Our testbed consisted of two computers, with one acting as the firewall, and the other acting as a packet generator. The firewall was a 2.4GHz Pentium 4 with 512MB RAM, with two 100Mbps Ethernet interfaces. The packet generator was a 700MHz Pentium III with 396MB RAM and a single 100Mbps Ethernet interface. Both computers ran Red Hat Linux 9. We connected the two computers by a cross-over Ethernet cable. The firewall's `eth1` interface was left unconnected (see Fig 8).

We configured the firewall's routing table to forward all the packets destined to the 10.0.0.0/8 class A subnet over the `eth1` interface to an imaginary next-hop router. Thus every incoming packet with a 10.*.*.* destination IP would pass through the `iptables` FORWARD chain. However, all the rules we generated had a DROP action, so no packets were actually forwarded—saving us the need to install a receiving host behind the firewall.

In each experimentation run, we loaded the firewall with randomly generated rules from the Perimeter model. We then let the packet generator send a sustained stream of packets, at a specified send rate, for a period of 10 seconds, after which it printed the exact number of packets it sent. All the packets were 80-byte TCP packets, with no TCP-flags set. After all the packets were sent, we recorded how many were filtered (and dropped) by `iptables`: `iptables` counts the number of packets that match each rule. If the send rate exceeds the firewall's maximal filtering rate, the firewall's IP buffers fill up, and packets start to drop — *before* they reach `iptables`. When this occurs, the total number of filtered packets reported by the `iptables` counters is *less* than the number of packets that were sent by the packet generator.

We verified that all the sent packets indeed arrived at the firewall computer, by sniffing its `eth0` interface using `ethereal`. Thus, all the packets that were lost, were lost on the firewall computer, within its IP layer. We did not encounter any layer-2 (Ethernet) loss. Note that even at 30,00pps, with 80-byte packets, the total bandwidth is only 19.2Mbps, which is easily sustainable on a dedicated 100Mbps link.

The packets we generated had random destination IP addresses in the range 10.0.*.*–10.7.255.255, random external source IP address, and TCP port numbers that were chosen according to an Internet mix [21]. In earlier simulations we

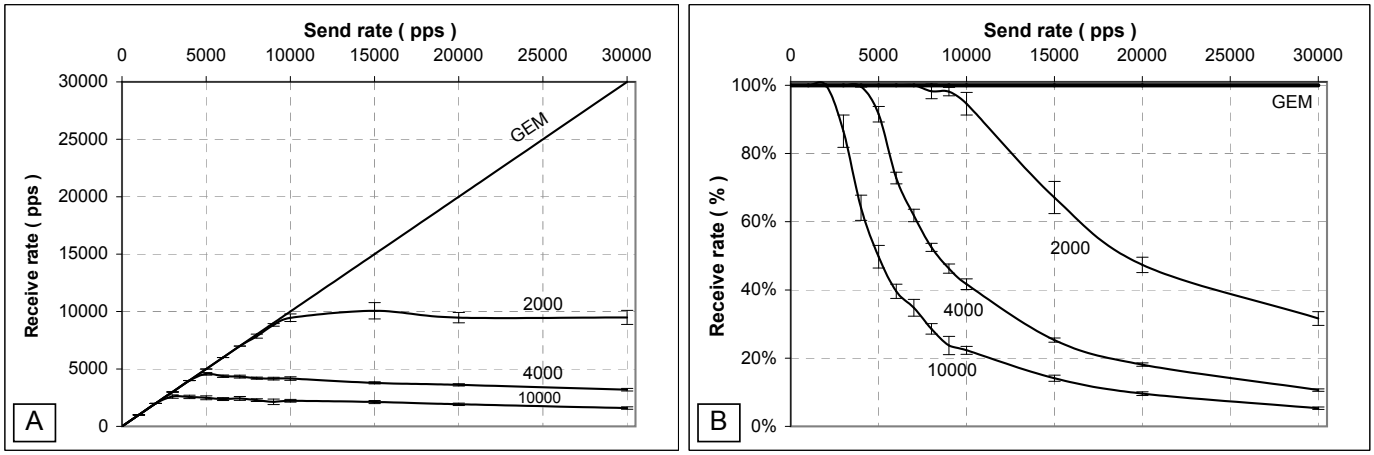


Fig. 9. Throughput of `iptables` with and without GEM, for different rule-base sizes. Figure A shows the receive rate as a function of the send rate, and figure B shows the throughput as a percentage.

verified that the firewall's matching speed is largely unaffected by the distribution of port numbers (both linear search and GEM). We omit the details.

Note that each packet mimics the first packet in a new TCP 3-way handshake—much like a SYN-flood DoS attack. This is reasonable for testing the performance of `iptables` because with a real TCP flow, any additional traffic on the same flow would have been matched by the fast state-lookup algorithm (i.e., by the `conntrack` module) and not by the "slow" `iptables` search algorithm.

### 5.2 Results and interpretation

We compared the matching throughput of `iptables` and `GEM-iptables` for rule-bases of 2000, 4000, and 10000 rules. The rules were created according to the distribution represented by the Inbound part of the Perimeter rules model (recall Table 3). For each rule-base size, we varied the packet send rate from 1000 pps up to 30,000 pps, and recorded the number of received (filtered by `iptables`) packets. The results can be seen in Fig 9. Every point on the curves is an average of 15 runs using three rule-bases of the given size. We also show the 90% confidence intervals.

Fig 9 clearly shows that `iptables` has a maximal throughput of between 2500 pps and 9000 pps (inversely proportional to the number of rules). This agrees with the results reported in [17] about the matching time of OpenBSD's `pf` [24], versus `iptables` and FreeBSD's `IPFilter` [26]. The reported maximal throughput in [17] was between 1500–3000 pps, for 1000 rules—but the author used a much slower machine than ours.

In contrast, GEM maintained a 100% throughput at all the send rates and for all rule-base sizes we tried. In fact, we were unable to reach send rates that cause GEM to lose packets. This is since the packet generating Perl script, running on the slower computer, hit a CPU bottleneck and could not send more than 30,000 pps. Thus we have not determined the maximal throughput of GEM, even with 10,000 rules. Based on the fact that the GEM search time only grows with the log of the number of rules, and on earlier simulation results (omitted), we extrapolate that GEM may well be able to filter at a rate of 100,000 pps.

### 5.3 Caveats

Besides matching on IP addresses, port numbers, and protocol fields, `iptables` also supports filtering based on other attributes of the packet, such as the IP fragmentation bit, TCP flags, the interface name, and rate limits. Currently GEM is unable to match such attributes. In the real firewall rule-bases (Section 3) that we checked, we did not encounter any rule that use this type of capability. Therefore we speculate that they are used rarely in typical firewall rule-bases.

There are several ways to handle an `iptables` rule-base which matches non-GEM attributes. One possibility is to add more dimensions to the GEM data structure. The obvious candidate would be the TCP flags field, that only has a handful of possible settings. Another possibility is to use a hybrid approach: Namely, we would need to split the rule-base into GEM rules and non-GEM rules. Every packet would then need to be filtered twice: once using GEM's efficient search, and once using a linear search over the non-GEM rules, giving two candidate winner rules. The winner rule would be the one with the lower rule number. Exploring these possibilities is left for future work.

## 6 Space Optimization Techniques

### 6.1 A Space-Time Trade-off

The GEM algorithm requires $O(n^d)$ space in the worst case, and has an $O(d \cdot \log n)$ search time complexity, where $d$ is the number of fields in packet header that are relevant for packet classification. In this section we suggest a trade-off, well-known in the computational geometry literature, which at the cost of a factor $\ell$ slowdown in the search time, provides an $\ell^{d-1}$ decrease in the space complexity. The next process describes the trade-off:

1) Split the firewall rule-base (arbitrarily) into $\ell$ sets of $n/\ell$ rules each. Append a final default "drop" rule to each partial rule-base, and give it a rule number of "infinity".

TABLE 4
Defining 2 and 3 parts splitting heuristics for perimeter model rule-base.

| | 2-part splitting rules |
|---|---|
| part 1 | all rules, which have $*$ in *source IP address* field |
| part 2 | all rules that not in part 1 |
| | 3-part splitting rules |
| part 1 | all rules, which have $*$ in *source IP address* field |
| part 2 | all rules that not in part 1 and have $*$ in *destination IP address* field |
| part 3 | all rules that not in part 1 and not in part 2 |

TABLE 5
Best field orders for heuristic splitting tests. The percentages indicate the fraction of rules in each part.

| | part 1 | part 2 | part 3 |
|---|---|---|---|
| 2 Parts | 0231<br>48.5% | 3021<br>51.5% | |
| 3 Parts | 0231<br>48.5% | 3120<br>43.8% | 3120<br>7.7% |

2) Build a GEM data structure for each partial rule-base separately. The size of each GEM-database will be $O((n/\ell)^d)$ in the worst case. The total size of the structure is:

$$O\left(\ell \cdot \left(\frac{n}{\ell}\right)^d\right) = O\left(\frac{n^d}{\ell^{d-1}}\right).$$

3) To match a packet header we have to match it against each of the $\ell$ GEM data structures. Each search contributes a matching rule for the packet. From these $\ell$ candidates we choose the one with the lowest number. Thus the overall search time complexity is $O(\ell \cdot \log \frac{n}{\ell} + \ell) = O(\ell \cdot \log \frac{n}{\ell})$.

Note that if we choose $\ell = O(n)$, then we get a GEM structure size of $O(n)$ and a linear search speed. At the other extreme, if we choose $\ell = 1$ we get the pure GEM complexity.

### 6.2 Evaluating the Effect of Splitting the Rule-Base

In order to evaluate the performance of the time-space tradeoff (Section 6.1), we experimented with the Perimeter model. We tried two splitting heuristics: The first heuristic is called '2-part', in which one part contains rules with source='$*$', and the other part contains all the other rules. In the other heuristic, called '3-part', the first part is the same as in 2-part splitting, the second part contains rules with destination='$*$' and source $\neq *$, and the third part is all other rules not included in parts 1 and 2.

#### 6.2.1 GEM Parts Information

Before we can proceed with the main test we have to determine the optimal orders for each part in both approaches. Table 5 shows that the best field order differs among parts: E.g., in part 1, the first field in the best order is the source IP (field 0). This is reasonable since all the rules in part 1 have source='$*$', so using it as the top-level field produces a single item in the second level and minimizes the size of the data structure.

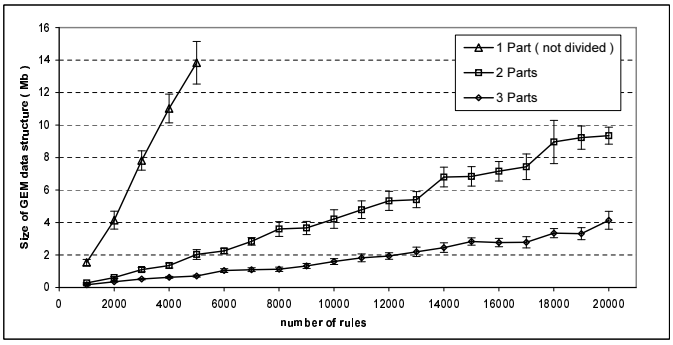


Fig. 10. GEM data structure size: unsplit, 2-parts splitting and 3-parts splitting.

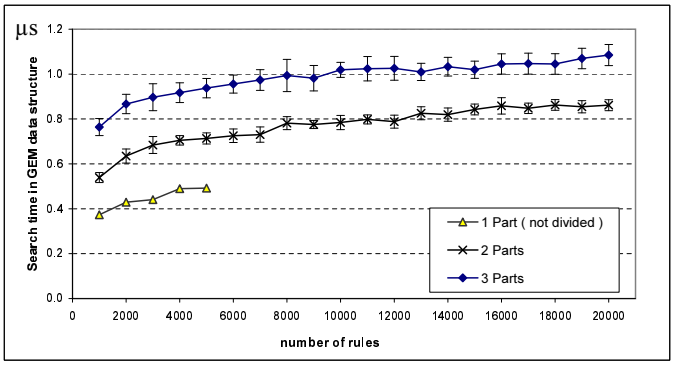


Fig. 11. GEM search time: unsplit, 2-part splitting, and 3-part splitting.

#### 6.2.2 GEM Data Structure Size

In order to test the build time, data structure size and search speed behavior, we generated rule-bases of sizes from 1000 to 20000 and built the GEM data structure using two approaches: 2-part heuristic splitting and 3-part heuristic splitting, as described above.

Fig 10 shows the data structure size of the unsplit, 2-part splitting, and 3-part splitting approaches. The figure clearly shows that both splitting heuristics are very effective in reducing the data structure size: The data structure size is reduced by a factor of 7 in the 2-part, and by a factor of 10 in the 3-part.

#### 6.2.3 GEM Search Time and Build Time

Fig 11 shows the search times for the different heuristics. We see that the theoretically expected results are true and that the search time is linear to the number of parts and is almost independent of the parts sizes.

An additional benefit from splitting is a significant reduction in build time for large rule-bases. For instance, building the 3-part GEM data structure for 20,000 rules takes about 10 sec, while the unsplit GEM data structure took over an hour to build.

## 7 Related Work

### 7.1 First match

The results closest to ours were presented by Gupta and McKeown in their Recursive Flow Classification (RFC) algo-

rithm [16]. They introduced an efficient packet classification algorithm, which is optimized for a hardware implementation. Their algorithm divides the address space into ranges created by borders of the rules, and encodes these ranges into a much smaller "number space". They then project the rules onto this smaller space, and repeat until the number space is small enough, at which point they assign the winning rule to each encoded range. The authors did not present an asymptotic time complexity analysis—however, based on our reading of their work we believe that RFC, like GEM, enjoys a logarithmic matching time, but suffers from an $O(n^d)$ worst-case space complexity, when the matching is performed on $d$ fields. By counting the machine instructions in their algorithm the authors claim that RFC should be able to process 1Mpps in isolation. The authors tested the actual space complexity on small-sized rule bases, provided by Internet Service Providers (ISPs), and claim that it grows linearly with number of rules. Interestingly, Gupta and McKeown remark that classical GEM-like algorithms from the field of computational geometry are applicable to the firewall matching problem—but they dismiss such algorithms as impractical due to their high (theoretical) space complexity. In contrast, our results show that on realistic rule-bases the space complexity of GEM grows linearly. Our simulations also show that, in isolation, GEM requires under 1$\mu$sec per packet and can handle well over 1Mpps. Finally, our emphasis is on a software implementation in the Linux kernel, and on very large rule-bases that are typical of enterprises rather than ISPs.

The work of [25] describes two algorithms: backtracking and set pruning tries. Both perform better than their respective theoretical bounds: $\Omega((\log n)^{d-1})$ time for backtracking and $O(n^d)$ space for set pruning tries. The authors used the field order to reduce the backtracking time, whereas we use the field order to reduce the required space. A survey of many packet classification algorithms implementing "first match" can be found in [36].

The work of Cohen and Lund [7], which appeared after our [28], offers a different approach using decision tree classifiers. Their construction uses linear space, yet has a sub-linear search time of $O(n^{0.63})$. Thus, their algorithms are significantly faster than the naive linear search, while still maintaining a linear space complexity. However, their algorithm is much slower than our logarithmic search time.

The GEM algorithm is a variant of the classical "slab method" algorithm of Dobkin and Lipton [10] for planar point location, which we adapted to the firewall domain. A survey of results in geometric range searching can be found in [22].

The algorithm of [11] uses a geometric approach (range queries and interval trees, cf. [9]), implements first-match semantics, and achieves logarithmic time matching, with near-linear space usage and a dynamic data structure that allows fast updates. However, this algorithm works in one dimension, and may be scaled to two dimensions, but it seems hard to extend to more than two dimensions.

Another algorithm, which uses a geometric approach, is the Area Based Quad-Trees (AQT) [4]. It has an $O((\log n)^{d-1})$ time complexity and allows fast updates.

In the field of computational geometry, [31] proposed an algorithm which solves the point location problem for $n$ *non-overlapping* $d$-dimensional hyper-rectangles, with a linear space requirement and $O((\log n)^{(d-1)})$ search time. In our case, we have *overlapping* $d$-dimensional hyper-rectangles, since firewall rules can, and often do, overlap each other—making rules overlap is the method firewall administrators use to implement intersection and difference operations on sets of IP addresses or port numbers. These overlapping hyper-rectangles can be decomposed into non-overlapping hyper-rectangles—however, a moment's reflection shows that the number of resulting non-overlapping hyper-rectangles is $\Omega(n^d)$, thus the worst case complexity of [31] for firewall rules is no better than that of GEM.

Note that [25], [4], [31], trade off search time for a linear space complexity. Our approach is to use the fastest possible search time ($O(\log n)$) - And we show that the penalty we suffer in the space complexity is still low enough to chose the GEM algorithm.

Interval Decision Diagrams were introduced by [6] as a tool for packet filtering using first-order logic. The idea is to construct a logic formula based on the integer intervals created by the set of rules. The algorithm enjoys logarithmic search time, but the build algorithm is exponential.

### 7.2 Longest prefix match

There is an extensive literature dealing with router packet matching, usually called "packet classification". Existing algorithms implement the "longest prefix match" semantics, using several different approaches.

The IPL algorithm of [12], which is based on results introduced in [20], divides the search space into elementary intervals by different prefixes for each dimension, and finds the best (longest) match for each such interval.

The Tuple Space Search algorithm is described in [33]. In this algorithm, all the prefixes are divided into tuples by field prefix length, and then searched linearly. To reduce the time complexity, the authors use pre-computations, markers and heuristic decisions based on statistics of tuples sizes. [32] introduced an extension to the Tuple Space Search algorithm that is optimized for hardware implementation.

Hash-based algorithms are proposed in [38], [35], [34]. These algorithms use hash tables for each prefix length and perform a binary search on those hash tables, coupled with various optimizations according to prefix statistics.

Other packet matching algorithms include Line Search on multi-dimensional tuple space [37], a modular approach with heuristic tree search [41], and two dimensional classification using prefix tuple space and different types of markers [39]. A survey of many packet matching algorithms implementing "longest prefix" semantics can be found in [15], [2], [1] and [30].

## 8 Conclusions and Future Work

We have seen that the GEM algorithm is an efficient and practical algorithm for firewall packet matching. We implemented it successfully in the Linux kernel, and tested its

packet-matching speeds on live traffic with realistic large rule-bases. GEM's matching speed is far better than the naive linear search, and it is able to increase the throughput of `iptables` by an order of magnitude. On rule-bases generated according to realistic statistics, GEM's space complexity is well within the capabilities of modern hardware. Thus we believe that GEM may be a good candidate for use in firewall matching engines.

We note that there are other algorithms that may well be candidates for software implementation in the kernel—specifically, we can point out the algorithms of Gupta and McKeown [16], Qiu et al. [25] and Woo [41]. We believe it should be quite interesting to implement all of these algorithms and to test them on equal footing, using the same hardware, rule-bases, and traffic load. Furthermore, it would be interesting to do this comparison with real rule-bases, in addition to synthetic Perimeter-model rules. We leave such a "bake-off" for future work.

As for GEM itself, we would like to explore the algorithm's behavior when using more than 4 fields, e.g., matching on the TCP flags, meta data, interfaces, etc. The main questions are: How best to encode the non-range fields? Will the space complexity still stay close to linear? What will be the best order of fields to achieve the best space complexity? Another direction to pursue is how GEM would perform with of IPv6, in which IP addresses have 128 bits.

## REFERENCES


[1] F. Baboescu, S. Singh, and G. Varghese, "Packet classification for core routers: Is there an alternative to cams," in *Proc. IEEE INFOCOM*, 2003.

[2] F. Baboescu and G. Varghese, "Scalable packet classification," in *Proc. ACM SIGCOMM*, 2001, pp. 199–210.

[3] N. Bar-Yosef and A. Wool, "Remote algorithmic complexity attacks against randomized hash tables," in *Proc. International Conference on Security and Cryptography (SECRYPT)*, Barcelona, Spain, Jul. 2007, pp. 117–124.

[4] M. M. Buddhikot, S. Suri, and M. Waldvogel, "Space decomposition techniques for fast Layer-4 switching," in *Protocols for High Speed Networks IV*, Aug. 1999, pp. 25–41.

[5] W. R. Cheswick, S. M. Bellovin, and A. Rubin, *Firewalls and Internet Security: Repelling the Wily Hacker*, 2nd ed. Addison-Wesley, 2003.

[6] M. Christiansen and E. Fleury, "Using interval decision diagrams for packet filtering," 2002, http://www.cs.auc.dk/~fleury/publications.html.

[7] E. Cohen and C. Lund, "Packet classification in large ISPs: Design and evaluation of decision tree classifiers," in *Proc. ACM SIGMETRICS*. New York, NY, USA: ACM Press, 2005, pp. 73–84.

[8] S. Crosby and D. Wallach, "Denial of service via algorithmic complexity attacks," in *Proceedings of the 12th USENIX Security Symposium*, August 2003, pp. 29–44.

[9] M. de Berg, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 2nd ed. Springer-Verlag, 2000.

[10] D. P. Dobkin and R. J. Lipton, "Multidimensional searching problems," *SIAM J. Comput.*, vol. 5, no. 2, pp. 181–186, 1976.

[11] D. Eppstein and S. Muthukrishnan, "Internet packet filter management and rectangle geometry," in *ACM-SIAM Symp. on Discrete Algorithms (SODA)*, 2001, pp. 827–835.

[12] A. Feldmann and S. Muthukrishnan, "Tradeoffs for packet classification," in *Proc. IEEE INFOCOM*, 2000, pp. 1193–1202.

[13] W. Feller, *An Introduction to Probability Theory and Its Applications*, 3rd ed. New York: John Wiley & Sons, 1967, vol. 1.

[14] "Firewall Wizards," Electronic mailing list, 1997–2009, archived at https://listserv.icsalabs.com/pipermail/firewall-wizards/.

[15] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, 2001.

[16] ——, "Packet classification on multiple fields," in *Proc. ACM SIGCOMM*, 1999, pp. 147–160.

[17] D. Hartmeier, "Design and performance of the OpenBSD stateful packet filter (pf)," in *Proc. FREENIX Track: 2002 USENIX Annual Technical Conference*, Jun. 2002.

[18] R. Jain, *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[19] S. Kandula, D. Katabi, M. Jacob, and A. Berger, "Botz-4-sale: Surviving organized DDOS attacks that mimic flash crowds," in *NSDI*, 2005.

[20] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proc. ACM SIGCOMM*, 1998, pp. 203–214.

[21] C. Logg and L. Cottrell, "Characterization of the traffic between SLAC and the Internet," March 2001, http://www.slac.stanford.edu/comp/net/netflow/SLAC-Netflow.html.

[22] J. Matoušek, "Geometric range searching," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 422–461, 1994.

[23] "The netfilter/iptables project, v1.2.7," 2002, http://www.netfilter.org/.

[24] "PF: OpenBSD packet filter," 2003, http://www.benzedrine.cx/pf.html.

[25] L. Qiu, G. Varghese, and S. Suri, "Fast firewall implementations for software and hardware-based routers," in *Proc. ACM SIGMETRICS*, 2001.

[26] D. Reed, "IP filter," 2003, http://coombs.anu.edu.au/~avalon/.

[27] D. Rovniagin and A. Wool, "The geometric efficient matching algorithm for firewalls," Dept. Electrical Engineering Systems, Tel Aviv University, Tech. Rep. EES2003-6, 2003, available from http://www.eng.tau.ac.il/~yash/ees2003-6.ps.

[28] ——, "The geometric efficient matching algorithm for firewalls," in *Proceedings of the 23th Convention of IEEE Israel*, Sep. 2004, pp. 153–156.

[29] G. Shwed, "System for securing inbound and outbound data packet flow in a computer network," U.S. Patent Number 5,606,668, Feb. 1997.

[30] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. ACM SIGCOMM*, 2003.

[31] M. Smid, "Dynamic rectangular point location with an application to the closest pair problem," *Information and Computation*, vol. 116, no. 1, pp. 1–9, Jan. 1995.

[32] V. Srinivasan, "A packet classification and filter management system," in *Proc. IEEE INFOCOM*, 2001, pp. 1464–1473.

[33] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *Proc. ACM SIGCOMM*, 1999, pp. 135–146.

[34] V. Srinivasan and G. Varghese, "Faster IP lookups using controlled prefix expansion," in *ACM Conference on Measurement and Modeling of Computer Systems*, 1998, pp. 1–10.

[35] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," in *Proc. ACM SIGCOMM*, 1998, pp. 191–202.

[36] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, 2005.

[37] M. Waldvogel, "Multi-dimensional prefix matching using line search," in *Proceedings of IEEE Local Computer Networks*, Tampa, FL, USA, Nov. 2000, pp. 200–207.

[38] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," in *Proc. ACM SIGCOMM*, September 1997, pp. 25–36.

[39] P. R. Warkhede, S. Suri, and G. Varghese, "Fast packet classification for two-dimensional conflict-free filters," in *Proc. IEEE INFOCOM*, 2001, pp. 1434–1443.

[40] D. D. Welch-Abernathy, *Essential Checkpoint Firewall-1: An Installation, Configuration, and Troubleshooting Guide*. Addison-Wesley, 2002.

[41] T. Y. C. Woo, "A modular approach to packet classification: Algorithms and results," in *Proc. IEEE INFOCOM*, 2000, pp. 1213–1222.

[42] A. Wool, "Architecting the Lumeta firewall analyzer," in *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., August 2001, pp. 85–97.

[43] ——, "Packet filtering and stateful firewalls," in *Handbook of Information Security*, H. Bidgoli, Ed. John Wiley & Sons, 2006, vol. III: Threats, Vulnerabilities, Prevention, Detection and Management, ch. 171, pp. 526–536.

[44] ——, "A quantitative study of firewall configuration errors." *IEEE Computer*, vol. 37, no. 6, pp. 62–67, 2004.