Enhancements to the Vantage Firewall Analyzer

Sandeep Bhatt, Prasad Rao
HP Laboratories
HPL-2007-154R1

Keyword(s):
firewall; rule set; overlap; analysis; rectangle intersection

Abstract:
The Vantage firewall analysis toolkit simplifies the complexity of managing firewall access control rule sets. Firewall rule sets typically become increasingly unwieldy over time. It is common for firewalls to have hundreds, or even thousands, of rules. As a result, administrators do not know how rules interact with each other. In a previous technical report [BHR], we presented our tool to analyze Checkpoint firewalls. Given two rule sets, the tool produces a comprehensive list of the traffic that one rule set will let through but not the other one. As such, we can use it to compare the existing rule set with a second rule set containing the proposed changes. The administrator can visually check if the difference in traffic patterns corresponds to what he or she intended in proposing the changes.

This report presents improvements and extensions to the toolkit. In particular, we present faster underlying algorithms and improved software architecture. We also extend the toolkit to analyze HP_UX IPFilter rule sets.

External Posting Date: June 7, 2008 [Fulltext] Approved for External Publication

Internal Posting Date: June 7, 2008 [Fulltext]

Enhancements to the Vantage Firewall Analyzer

Sandeep Bhatt, Prasad Rao
{sandeep.bhatt, prasad.rao}@hp.com
Trusted Systems Laboratory
5 Vaughn Drive
Princeton, NJ 08540

Abstract

The Vantage firewall analysis toolkit simplifies the complexity of managing firewall access control rule sets. Firewall rule sets typically become increasingly unwieldy over time. It is common for firewalls to have hundreds, or even thousands, of rules. As a result, administrators do not know how rules interact with each other. In a previous technical report [BHR], we presented our tool to analyze Checkpoint firewalls. Given two rule sets, the tool produces a comprehensive list of the traffic that one rule set will let through but not the other one. As such, we can use it to compare the existing rule set with a second rule set containing the proposed changes. The administrator can visually check if the difference in traffic patterns corresponds to what he or she intended in proposing the changes.

This report presents improvements and extensions to the toolkit. In particular, we present faster underlying algorithms and improved software architecture. We also extend the toolkit to analyze HP_UX IPFilter rule sets.

1 Introduction

Perimeter firewalls are the first line of defense for enterprise networks. Used internally, they isolate subnets with different security profiles. Large enterprises impose complex patterns of traffic flows among different divisions and organizations, and for external access from the public internet and from supply-chain partner networks. Firewalls manage the corresponding information flows using a rule set. Individual rules within a rule set determine which packets can pass through a firewall and which are blocked. The design and ongoing maintenance of the firewall rule sets is of critical importance to enterprise security.

There are many reasons for changing firewall rule sets. Business changes such as the creation or dissolution of business partnerships, and internal organizational changes require changes in allowed traffic patterns. New security-conscious standards often disallow previously allowed telnet and ftp connections or outgoing VPN connections. Finally, changes in technology require changes in traffic patterns – for example, new vulnerabilities and advisories, patching mechanisms, vendor changes and upgrades, application changes such as port or protocol changes from JMS to Web Services, and transport changes from leased lines to cheaper SSL VPN connections.

The result of all these changes over extended periods is that the number of rules in the rule set becomes large and unwieldy. It is common for large enterprises to have firewalls

with hundreds of rules. Making sense of the rules, and especially of the interaction between different rules, poses a challenge. Often, IT organizations are wary of making changes to perimeter firewalls for fear that the change may either block business-critical traffic or else open a new vulnerability. Further aggravating the management problem is the churn in firewall administrators – all too often, the understanding of the subtle interactions between rules in a rule set resides only with the administrator. As administrators change, the rationale behind the rule set is easily lost.

This report extends the toolkit described in our previous report [BHR] in several ways. First, we present faster algorithms to analyze rule sets; the new algorithm scales efficiently to large rule sets, and we report results on rule sets with thousands of rules, provided by HP Outsourcing Services. Second, we report improvements in the software architecture of the toolkit. Finally, we have extended the toolkit to analyze HP_UX IPFilter rule sets.

2 Prototype Background

In this section, we briefly review the features supported by the Vantage firewall analysis toolkit, reported in [BHR].

The inputs to the tool consist of two Checkpoint configuration files: objects.c and rules.w. The first file contains definitions of network objects, *N*, while the second contains the rule set, *A*, on network objects and services. The tool also accepts as input a file containing a log of historical traffic from the Checkpoint firewall. The time span of the logs *L* may cover times during which the firewall enforced different rule sets.

The prototype supports the following features:

- For a specified packet *p* determine if *A* will block *p* or allow it to pass through.
- Determine, for every rule in *A*, whether one or more rules of higher priority *eclipse it* by capturing some of the packets it is designed for.
- Given a second rule set *B*, list all the traffic patterns that will be blocked by *A* but not by *B* and vice versa.
- Determine which packets in the logs *L* were blocked by the firewall, but which the rule set *A* would let through, and vice versa.

The first feature simply mimics the firewall – the tool informs the user which packets will be blocked by the firewall and which will pass through.

The second feature comprehensively analyzes the rule set: for every rule, the tool determines if the higher-priority rules override, either some part or the entire effect, of any given rule. This determines redundant, or dead, rules that the administrator can remove to manage the complexity of the rule set.

The third feature computes the difference in allowed traffic between two different rule sets. This allows the user to check the existing rule set against a set of proposed changes *before deploying the changes to the rule set.* This is a useful feature for planning and administering rule set changes to complex firewalls.

The final feature allows a user to examine how historical traffic would have been treated had the given rule set been deployed throughout the time span of the logs.

3 Technical Background

A Checkpoint NG firewall rule can be represented simply as

<id, Source, Destination, Service, Action>,

where *id* is a unique integer identifier for the rule, *Source, Destination* and *Service* are defined in the file named objects.c, and *Action* is either permit or deny.[1] The named source and destination fields are specified as sets of IP address ranges. For example, the name *research* may be defined as the set of IP addresses 192.168.1.0/8 and 192.168.3.0/8. Similarly, the objects.c file maps named services to sets of pairs of protocol and port ranges. For ease of exposition, we will assume that each source and destination maps to one range of IP addresses, and that each service maps to one protocol and port range pair.

A firewall rule determines whether packets for a certain kind of *service* from one IP address range to another IP address range may pass through the firewall. A rule applies to a packet if the source, destination and service specified in the packet match those in the rule; if so, the action field determines if the packet is to be allowed through, or dropped.

Now, consider the following two rules:

```
1. 192.168.1.*   10.1.1.1   tcp   allow
2. 192.168.1.1   10.1.1.*   tcp   deny
```

If the packet is of the form <192.168.1.1, 10.1.1.1, tcp> then both rules match the packet. When such a conflict occurs, the rule with the higher priority applies. Like many other firewalls, Checkpoint NG processes rules in linear order – without loss of generality, we assume that rules are ordered by their *id* field with a lower number signifying higher priority. In our example, the first rule thus takes priority over the second, and the packet will pass through. If we were to switch the order of the rules, however, the packet will be blocked.

Next, a packet of the form <192.168.1.2, 10.1.1.1, tcp> matches the first rule, but not the second. Consequently, it will pass through, no matter how the two rules are ordered.

This simple example demonstrates that the ordering of rules changes firewall behavior when rule definitions overlap on their source, destination and service specifications. When a firewall rule set contains hundreds, or even thousands, of rules it is very likely that rule definitions will overlap in complex ways such that: (i) a particular rule will never apply because it is eclipsed by higher priority rules, or (ii) unintended traffic flows through the firewall, or else is inadvertently blocked.

[1] We omit other fields in the checkpoint firewall such as through, documentation strings, time etc for clarity.

Overlapping rules in a large rule set are especially problematic for firewall administrators to manage. When the administrator adds a new rule, it may not be easy to see if previous rules eclipse the new rule, whether partially or completely, rendering the new rule either partially effective or redundant. Moreover, the new rule might itself eclipse subsequent rules in the rule set, thus rendering them either partially or totally ineffective. Similarly, deleting a rule from the rule set can have unintended and undesirable effects.

Native firewall analysis tools provide some help. Checkpoint provides a user interface with its firewall that identifies when a rule totally (but not partially) eclipses another. In addition, most firewall rules provide some type of *testing capability* to test the rule lists against handcrafted packets. However, such exhaustive testing places the burden of creating the test packets on the administrator, who might easily overlook some consequential test cases.

With this background, this report addresses the following technical questions:

Given an ACL, compute all the interactions between rules; specifically, for each rule indicate whether higher priority rules eclipse it, either partially or totally).

Given two ACLs, determine the difference in allowed traffic; specifically, list all traffic patterns that one firewall allows but the other one blocks.

We address these questions in the next two sections. For further details on log-analysis, we refer the interested reader to our previous report [].

4 An efficient algorithm to detect rule overlaps

The main idea is to process the rule set and build a data structure that facilitates quick detection of overlaps among rules. We represent an IP address as a 32-bit integer, so that an IP address range (IP address plus a subnet mask) corresponds to an integer interval. We restrict attention to IP address ranges represented by a *k*-bit prefix (implicitly followed by 32 – *k* wild card bits); the end points of the interval are obtained by setting the wild card bits to all zeros or all ones respectively.[2]

Similarly, we map service definitions (set of port ranges) to integers so that a service corresponds to a set of line segments. With this, we establish a correspondence between a rule <*Source, Destination, Service*> and a set of three-dimensional regions (with *x* and *y* coordinates for IP ranges, and *z* coordinate for port ranges). A firewall rule refers to packets that fall within the set of cubes corresponding to that rule.

For ease of exposition, we focus on a single service, so that each rule corresponds to a two-dimensional rectangle. With this restriction, two rules overlap if and only if the corresponding rectangles intersect. This reduces the problem of finding all rule overlaps to finding all the intersections within a set of two-dimensional rectangles.

[2] If a source or destination set is not defined by a prefix, we can split the address range into a set of prefix addressable ranges. In principle, this can increase the number of addresses by a logarithmic factor. An alternative for general address ranges is to use an interval tree.

The remainder of this section sketches the algorithms for finding overlaps, and for computing the difference between two access control lists. We highlight the salient points, leaving out the tedious implementation details.

4.1 Finding all rectangle intersections

Given a set of *n* two-dimensional rectangles, we present an O(*nlogn* + *k*) time algorithm to compute all *k* intersections.

Input: set of *n* 2*d*-rectangles, each represented as a pair (*x*,*y*) of the source and destination IP prefix address ranges.
Output: all pairs of intersecting rectangles.

Sketch of the algorithm:

Step 1. Sort the *x*-coordinates of all rectangles - call this sorted list $x_1,x_2,....,x_{2n}$.

Step 2. Consider a vertical line that sweeps across the *x*-axis. An "event" occurs when the sweep line encounters one of the end-points x_i above. Initially, all rectangles are "asleep;" when the sweep line encounters the left-edge of a rectangle it becomes active, and when the sweep line encounters its right end it becomes inactive. We will maintain a data structure containing all active rectangles. Figure 1 illustrates the process.

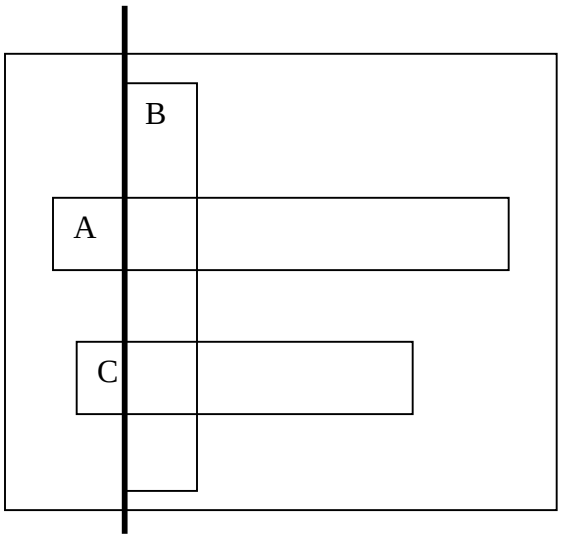

Figure 1. When the sweep line (shown in bold) encounters the left edge of rectangle B, rectangles A and C are active.

Step 3. When a rectangle first becomes active, its candidates for intersections are the currently active rectangles. To find the actual intersections we need to find all active

rectangles whose *y*-span intersects with the *y*-span of the newly active rectangle. The intersections found are stored in a list that is output upon termination.

The algorithm terminates when the sweep line crosses x_{2n}. At this point, the output list contains all pairs of intersecting rectangles.

To complete the algorithm outline, we need a data structure that stores active rectangles and supports the following operations:

1. Insert rectangle
2. Delete rectangle
3. Given an address range *r*, find all active rectangles whose *y*-span intersects *r*.

For general rectangles, an interval tree data structure [] insertion and deletion in time O(log *n*), and finds intersecting rectangles in time O(*s* + log *n*) where *s* is the number of intersections for the query interval. While theoretically optimal, interval trees are complex to implement. Fortunately, there is a simpler option.

The key observation is that two distinct prefix ranges *I* and *J* can intersect if and only if one is contained within the other. We store all the active rectangles in a trie-structure with *y* coordinate as key. As an optimization, we compress chains of degree-two nodes so that every internal node has exactly two children.

All intersecting active rectangles for range *r* can be found as follows. Locate the trie node representing *r*; all active rectangles in the path from *r* to the root of the trie, or in the sub-trie of *r* are exactly the set of active rectangles that intersect *r*. Figure 2 illustrates this with an example.

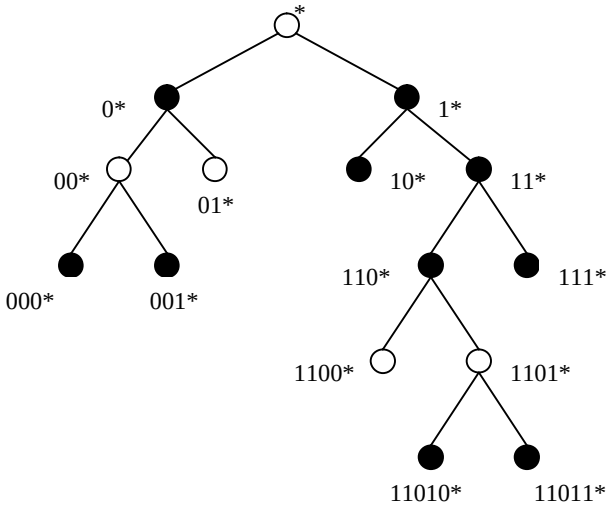

Figure 2. Trie containing active rectangles in black. The rectangles intersecting 110* include 1*, 11*, 11010* and 11011*.

5 Rule set decomposition

In this section, we modify the previous algorithm to compute a decomposition of the two-dimensional IP address space for a service, into disjoint rectangular regions that cover all packets that are allowed through the firewall by the rule set.

Input: set of n 2d-rectangles, each represented as a tuple (x,y,d,p) of the source and destination IP prefix address ranges, a decision d which is either "allow" or "deny," and a distinct priority p.
Output: for each service, a partition of the IP address space into a set of disjoint rectangular regions, each labeled "allow" or "deny," with every "allow" region labeled by a priority. Every IP packet that is allowed through the firewall must lie in a region labeled "allow" and must trigger the firewall rule with the priority of the region. All packets blocked by the firewall must lie within a deny region.

Sketch of the algorithm:

Step 1. Sort the x-coordinates of all rectangles - call this sorted list x_1,x_2,....,x_{2n}. We will refer to this as the list of "pending" rectangles.

Step 2. Consider a vertical line that sweeps across the x-axis. We will maintain a data structure to store the set of "active" disjoint rectangular regions that are carried by the sweep line; as the sweep line moves, these regions will be extended, terminated or split into smaller disjoint regions.

Initially, the set of active regions consists of the entire IP address space and is marked the lowest priority deny, in accordance with the default deny rule (or else it is marked allow in case of a default allow rule). When the sweep line encounters the left-edge of a rectangle, the rectangle becomes active, and when the sweep line encounters its right end it becomes inactive. The active regions are updated as described below.

Let N be the new pending rectangle encountered by the sweep line. We first find all the active regions that intersect the y-span of the new pending rectangle. We process these active regions one-by-one. Let R be an intersecting active region. There are a number of cases to analyze, depending on the decision and priority labels on N and R.

In what follows, we describe some example cases; we leave an exhaustive analysis of all cases to the implementation. In our example, suppose that both N and R are marked allow. There are four different ways the rectangles can intersect, as shown in Figure 3 below.

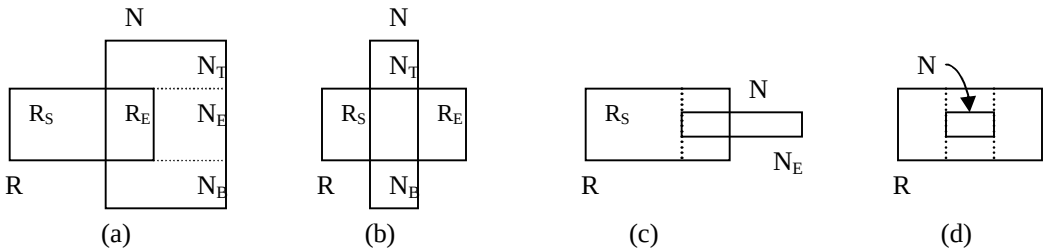

Figure 3. *R* is an active region, *N* is the new rectangle encountered by the sweep line and both are marked "allow."

Consider the case that *N* has greater priority than *R*. The actions in each case would be as follows.

a. Output R_S and delete R from the active regions.
b. Output R_S, delete R, and insert R_E to the active regions
c. Output R_S, delete R, and insert R_T, and R_B to the active regions
d. Output R_S, delete R, and insert R_T, R_B and R_E to the active regions

If the priority of *N* were less than the priority of *R*, and both were allow rectangles, the actions for the four cases would have been as follows.

a. Remove *N* from the pending rectangles, and replace it with N_T, N_B, and N_E.
b. Remove *N* from the pending rectangles, and replace it with N_T and N_B.
c. Remove *N* from the pending rectangles, and replace it with N_E.
d. Remove *N*.

As each active region that intersects *N* is processed, *N* may be carved up into smaller pieces. When every active region that intersects *N* has been processed, the portions of *N* that remain are inserted into the list of active regions and/or the list of pending rectangles, as appropriate.

The remaining cases are all similar, and it is straightforward to prove that when all pending rectangles have been processed, that the output list partitions the IP address space into rectangular regions, each labeled either deny or allow according to whether the rule set blocks or allows all packets within that rectangle to flow through the firewall. We leave details to the interested reader.

6 Comparing two rule sets

Once we have partitioned the IP address space into allow and deny rectangles, we can check for the equivalence of two rule sets.

Input: Two rule sets *A* and *B*.

Output: A set of rectangles containing all the packets for which the decisions of *A* and *B* are different.

Sketch of the algorithm:

Step 1. Compute the two decompositions, one for *A* and the other for *B*.

Step 2. Sort the *x*-coordinates of all the regions and sweep a vertical line across the *x*-axis. At each step, we will maintain a set of disjoint intervals that covers all packets along the sweep line for which the outcomes of the two firewalls differ. The intervals are initialized for x=0.

Step 3. As the sweep line progresses, these intervals define rectangular regions in which the firewalls behave differently. Figures 4 and 5 illustrate with an example. When the sweep line encounters a new rectangle, the rectangles defined by the disjoint intervals are output, and the intervals are updated for the new position of the sweep line. This step is repeated until the sweep line encounters the boundary of the IP address space.

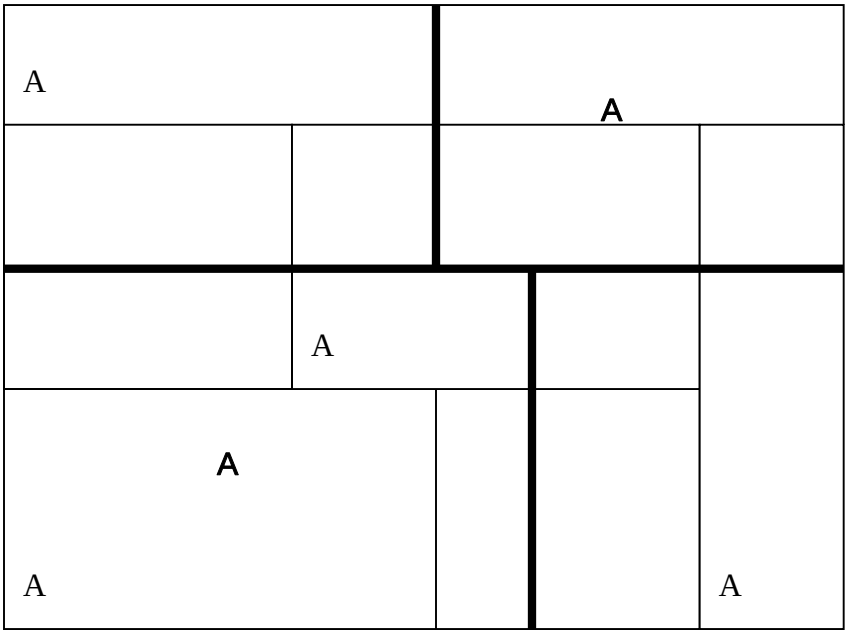

Figure 4. Two decompositions defined by the two rule sets. The allow regions are marked A; the unmarked regions are deny regions. The bold lines define 4 regions for A, while B is split into 6 regions of which 4 are allow regions.

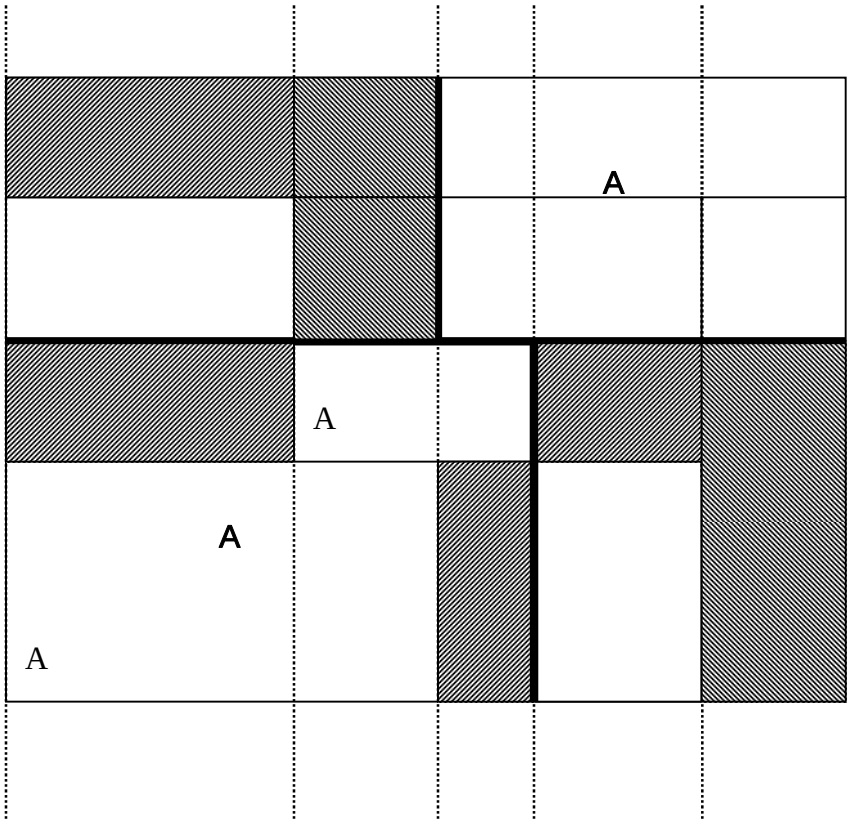

Figure 5. Each position of the sweep line is split into intervals where the two rule sets differ. As the sweep line moves to the right, each interval sweeps a shaded rectangle where the two rule sets differ. A new decomposition of the sweep line is computed every time a new region is encountered.

7 Analyzing IPFilter Rule sets

In this section, we describe how to analyze IPFilter rule sets. IPFilter rules have a different syntactic structure, and the firewall interprets the rule set in a different way than Checkpoint does. IPFilter rules apply to individual interfaces; if a firewall has *k* interfaces then the rules are converted into 2*k* rule sets: an incoming and an outgoing rule set for each interface, and each of these rule sets are individually analyzed using our tool.

As we show below, we can convert an IPFilter rule set into an equivalent Checkpoint rule set by reordering the rules and making simple transformations on individual rules. To analyze overlaps within an IPFilter rule set we first convert the rule set into an equivalent Checkpoint rule set, and use our analyzer. We can similarly compare two IPFilter rule sets, by converting each rule set, and comparing the resultant rule sets.

The rest of this section briefly introduces IPFilter rules and describes the conversion process.

7.1 IPFilter rules

IPFilter processes the rules in top-down order, but unlike Checkpoint, uses the last-match principle. This means that even after finding a match, the IPFilter engine must scan the remaining rules. Now, if this were the only difference, we could simply reverse the rule set, apply the first-match principle, and the outcome would be equivalent.

However, IPFilter rules use a few additional keywords that modify last-match. In addition to the five fields *id, Source, Destination, Service,* and *Action*, that we have discussed thus far, IPFilter rules use the following additional keywords *quick, head,* and *group.*[3] We explain the meaning of these keywords next.

In processing the rules sequentially, if IPFilter encounters a rule that matches the packet and the rule contains the keyword *quick*, then the rule wins, and IPFilter does not examine the remaining rules. This effectively overrides last-match.

There is a simple way to reorder a rule set *A* with *quick* rules into a rule set *B* so that first-match processing on *B* (with the quick keywords removed) is equivalent to last-match processing on *A*. We transform *A* as follows: suppose that it has *k* quick rules and *n-k* non-quick rules. Create a rule set *A'* in which the first *k* rules are the quick rules from *A*, in the same order they appear in *A* but with the keyword removed. To this append the remaining *n-k* non-quick rules *A*, but in *reverse order of appearance in A.* Call this transformed rule set *B.*

[3] We are not faithful to the precise syntax of IPFilter rules; however, these fields capture the essential concepts. The parts that we omit do not significantly alter our analysis.

To see that first-match processing on *B* is equivalent to last-match processing on *A*, we consider a packet *p* and argue that the same rule wins in both rule sets. We consider two cases. Suppose that the winning rule in *A* is a quick rule *r*. We argue that *r* will win during first-match processing on *B*; this is true because the quick rules in *B* that precede *r* cannot match *p*, otherwise *r* would not have won in *A*. Next, if the winning rule *r* in *A* is a non-quick rule, then none of the quick rules can match *p*; by our previous argument, first-match processing on the reversed rule set is equivalent to last-match processing on the original rule set, and therefore the winning rule in *B* will be *r*.

In addition to *quick*, IPFilter uses a pair of related keywords *group* and *head*, which we examine, in the next section.

7.2 IPFilter groups

The keyword *group* allows rules to be organized hierarchically in a tree structure, instead of as a linear list, so that if an incoming packet is unrelated to a set of rules, these rules will not be processed. This reduces IPFilter processing time on each packet and improves system performance.

An example of a nested group structure is given in Table 1 below. The first rule in a group contains the keyword *head* and specifies the group number. If a packet does not match the head of a group, then the rules in the group are skipped and processing continues at the current level. If the packet does match the head, the group rules are examined. If the head rule contains the *quick* keyword, then the packet processing is restricted to the group rules and does not return to the parent level.

It is convenient to visualize hierarchically organized rules in a tree structure. For example, Figure 6 is a tree representation of the rules in Table 1. Each node in the tree is labeled by the corresponding set of rules; nodes colored black correspond to *quick* rules. Head rules are not colored; the keyword *quick* on a head rule means that the entire group is labeled *quick*. In this case we create dummy shaded nodes to mark the group as a quick group.

To obtain an equivalent first-match rule set, we use a recursive strategy. To linearly order the subtree rooted at a node, first recursively linearize the children of that node that are shaded (quick groups) in the order they appear (left first). Next, append the linear orders of the remaining children, but in reverse (right first) order.

```
 0: block in quick  on lan0 all head 1
 1: pass  in on lan0 from 192.169.0.0 to any group 1
 2: block in quick on lan0 from 192.168.0.0/16 to any group 1
 3: block in quick on lan0 from 172.16.0.0/12 to any group 1
 4: block in quick on lan0 from 10.0.0.0/8 to any group 1
 5: block in quick on lan0 from 127.0.0.0/8 to any group 1
 6: block in log quick on lan0 from 20.20.20.0/24 to any group 1
 7: block in log quick on lan0 from any to 20.20.20.0/32 group 1
 8: block in log quick on lan0 from any to 20.20.20.63/32 group 1
 9: block in log quick on lan0 from any to 20.20.20.64/32 group 1
10: block in log quick on lan0 from any to 20.20.20.127/32 group 1
11: block in log quick on lan0 from any to 20.20.20.128/32 group 1
12: block in log quick on lan0 from any to 20.20.20.255/32 group 1
13: block in quick on lan0 from 21.20.20.0/24 to any group 1
14: block in log  on lan0 from any to 21.20.20.0/32 group 1
15: block in log  on lan0 from any to 21.20.20.63/32 group 1
16: block in log quick on lan0 from any to 21.20.20.64/32 group 1
17: block in log  on lan0 from any to 21.20.20.127/32 group 1
18: block in log  on lan0 from any to 21.20.20.128/32 group 1
19: block in log  on lan0 from any to 21.20.20.255/32 group 1
20: pass in log on lan0 from any to 22.20.20.0/24 group 1 head 11
21: pass in log  on lan0 from any to 22.20.20.0/32 group 11
22: pass in quick log  on lan0 from any to 22.20.20.63/32 group 11
23: pass in log quick on lan0 from any to 22.20.20.64/32 group 11
24: block in quick log  on lan0 from any to 22.20.20.127/32 group 11
25: pass in log  on lan0 from any to 22.20.20.128/32 group 11
26: pass in log  on lan0 from any to 22.20.20.255/32 group 11
27: pass  in on lan0 all group 1
28: pass out on lan0 all
29: block out quick on lan1 all head 10
30: pass out quick on lan1 proto tcp from any to 20.20.20.64/26  port = 80 flags S keep state group 10
31: pass out quick on lan1 proto tcp from any to 20.20.20.64/26  port = 21 flags S keep state group 10
32: pass out quick on lan1 proto tcp from any to 20.20.20.64/26  port = 20 flags S keep state group 10
33: pass out on lan1 proto tcp from any to 20.20.20.65/32  port = 53 flags S keep state group 10
34: pass out on lan1 proto udp from any to 20.20.20.65/32  port = 53 keep state group 10
35: pass out on lan1 proto tcp from any to 20.20.20.66/32  port = 53 flags S keep state group 10
36: pass out on lan1 proto udp from any to 20.20.20.66/32  port = 53 keep state group 10
37: pass in proto tcp from 1.0.0.0-9.0.0.0 to any port = 23 keep state head 101
38: pass in proto tcp from 2.0.0.0-8.0.0.0 to any port = 23 keep state head 102 group 101
39: pass in proto tcp from 3.0.0.0-7.0.0.0 to any port = 23 keep state head 103 group 102
40: pass in proto tcp from 4.0.0.0-6.0.0.0 to any port = 23 keep state head 104 group 103
41: pass in proto tcp from 5.0.0.0-5.5.0.0 to any port = 23 keep state group 104
```

Table 1. A sample IPFilter Rule Set.

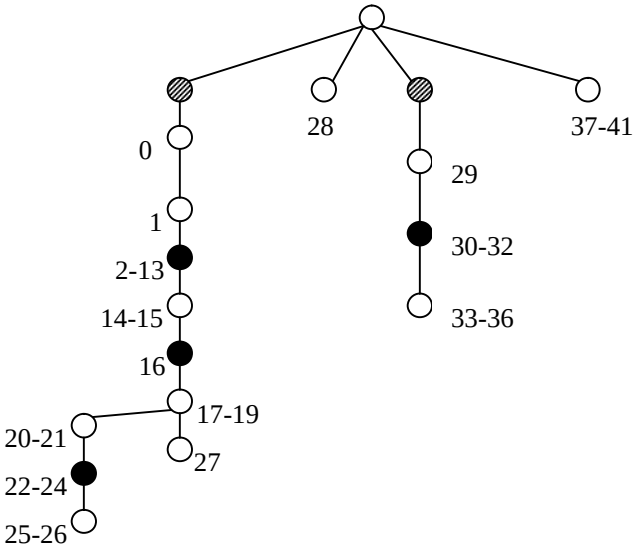

Figure 6. Tree representation of the rule set of Table 1. Black nodes correspond to quick rules, and shaded nodes correspond to quick groups.

```
 2: block in quick on lan0 from 192.168.0.0/16 to any group 1
 3: block in quick on lan0 from 172.16.0.0/12 to any group 1
 4: block in quick on lan0 from 10.0.0.0/8 to any group 1
 5: block in quick on lan0 from 127.0.0.0/8 to any group 1
 6: block in log quick on lan0 from 20.20.20.0/24 to any group 1
 7: block in log quick on lan0 from any to 20.20.20.0/32 group 1
 8: block in log quick on lan0 from any to 20.20.20.63/32 group 1
 9: block in log quick on lan0 from any to 20.20.20.64/32 group 1
10: block in log quick on lan0 from any to 20.20.20.127/32 group 1
11: block in log quick on lan0 from any to 20.20.20.128/32 group 1
12: block in log quick on lan0 from any to 20.20.20.255/32 group 1
13: block in quick on lan0 from 21.20.20.0/24 to any group 1
16: block in log quick on lan0 from any to 21.20.20.64/32 group 1
22: pass in quick log on lan0 from any to 22.20.20.63/32 group 11
23: pass in log quick on lan0 from any to 22.20.20.64/32 group 11
24: block in quick log on lan0 from any to 22.20.20.127/32 group 11
27: pass in on lan0 all group 1
26: pass in log on lan0 from any to 22.20.20.255/32 group 11
25: pass in log on lan0 from any to 22.20.20.128/32 group 11
21: pass in log on lan0 from any to 22.20.20.0/32 group 11
20: pass in log on lan0 from any to 22.20.20.0/24 group 1 head 11
19: block in log on lan0 from any to 21.20.20.255/32 group 1
18: block in log on lan0 from any to 21.20.20.128/32 group 1
17: block in log on lan0 from any to 21.20.20.127/32 group 1
15: block in log on lan0 from any to 21.20.20.63/32 group 1
14: block in log on lan0 from any to 21.20.20.0/32 group 1
 1: pass in on lan0 from 192.169.0.0 to any group 1
 0: block in quick on lan0 all head 1
30: pass out quick on lan1 proto tcp from any to 20.20.20.64/26 port = 80 flags S keep state group 10
31: pass out quick on lan1 proto tcp from any to 20.20.20.64/26 port = 21 flags S keep state group 10
32: pass out quick on lan1 proto tcp from any to 20.20.20.64/26 port = 20 flags S keep state group 10
36: pass out on lan1 proto udp from any to 20.20.20.66/32 port = 53 keep state group 10
35: pass out on lan1 proto tcp from any to 20.20.20.66/32 port = 53 flags S keep state group 10
34: pass out on lan1 proto udp from any to 20.20.20.65/32 port = 53 keep state group 10
33: pass out on lan1 proto tcp from any to 20.20.20.65/32 port = 53 flags S keep state group 10
29: block out quick on lan1 all head 10
41: pass in proto tcp from 5.0.0.0-5.5.0.0 to any port = 23 keep state group 104
40: pass in proto tcp from 4.0.0.0-6.0.0.0 to any port = 23 keep state head 104 group 103
39: pass in proto tcp from 3.0.0.0-7.0.0.0 to any port = 23 keep state head 103 group 102
38: pass in proto tcp from 2.0.0.0-8.0.0.0 to any port = 23 keep state head 102 group 101
37: pass in proto tcp from 1.0.0.0-9.0.0.0 to any port = 23 keep state head 101
28: pass out on lan0 all
```

Table 2. Rule set of Table 1 transformed into equivalent first-match

8 Prototype

This section describes the operation of the firewall analyzer for checkpoint data sets. The processing of other firewall rule sets is similar, but the details slightly different.

For checkpoint, the input consists of an *objects* file and a *rules file.* The object file contains definitions of the network and service objects and the *rules* file contains the rules that use these objects.

First, the inputs are parsed into an abstract syntax tree. Much of the stuff in the tree is irrelevant for the analysis, and is pruned away in order to avoid consuming too much memory.

The parser populates the network and service objects needed by the analyzer by querying the tree. Finally, the parser releases references to the tree – to make it available to the garbage collector.

The analyzers converts the three dimensional problem of analyzing rule sets into a two-dimensional one, by analyzing the rules a service at a time. The analyzer generates and stores results of the analysis on the file system, and starts a web server that enables the user to drill down into the results using a browser.

To summarize: the system consists of the following components

1. The analyzer: Analyzes the data structures and replies to queries
2. The parser: parses the rules, object files of the firewall (checkpoint for our prototype), and feeds it to the analyzer.
3. The server: Runs a jetty web server and displays results on a browser. Also presents forms for query and comparing impacts.
4. The persistence layer stores intermediate results for analysis, and we have currently implemented it using files. We also have an alternative implementation that uses any jdbc compliant database.

In the current version of the prototype the components are implemented in java 1.5 for ease of installation and use. The previous (currently retired) version of the prototype was in python – for ease of programming.

We have had encouraging results from this prototype on real production data sets with up to seven thousand rules. (This seven thousand rule data set actually corresponded to eight hundred and thirty one independent rule sets each with at most a couple of hundred rules). The analyzer itself is very fast (the analysis and comparison of rule sets, not including the time to print the results in html files, runs in under a second on all our tests). We have traced most of the delay experienced in the user interface to the act of writing html files.

Our algorithms are inherently scalable, so we do not expect any difficulties in analyzing much larger data sets.

9 Extensions

The prototype is currently specific to Checkpoint (both Firewall 1 and NG). We can extend the current prototype in the following directions.

1. Extend the analysis to cover firewall products from different vendors. As mentioned earlier, this will require developing a parser for each product (to accommodate its native rule set and object definition formats). The difficulty of writing a parser ranges from easy for XML based configuration files such as in ISA server, to moderate for configurations such as CISCO and NetScreen.
2. Extend the tool to compare two different rule sets for two different firewall products. This will help administrators planning a transition from one firewall type to another.
3. Given a rule set for one type of firewall, produce a rule set for a second type of firewall, such that the allowed traffic flowing through both firewalls is identical.
4. Given a rule set for a firewall, minimize the number of rules without changing the allowed traffic.

10 Related Work

There has been a flurry of work on firewall analysis in recent years. The most relevant is the Firewall Analyzer (FA) product from Algorithmic Security Inc [ALG] that builds on previous research reported in [MWZ] and [WOO]. FA analyzes a number of different types of firewalls, reports common vulnerabilities found in the rule set, and detects rules that are redundant (i.e., are eclipsed by higher-priority rules). FA does not check equivalence of rule sets.

The Solsoft Firewall Manager [SOL] converts high-level specifications of allowed and prohibited traffic into firewall rule sets. However, it does not analyze existing rule sets on a firewall to see if they comply with the high-level specifications.

There are also a number of academic papers on firewall analysis. The paper [AH1] recognizes different types of conflicts that can occur between a pair of rules, but does not address the more general problem of one rule being eclipsed by a set of rules. The authors extend their study to rules on multiple firewalls in a tree network [AHB], but the results are again limited to interactions between pairs of rules. The paper [GLI] proposes a simplistic way to design rule sets such that every packet is associated with exactly one rule; the problem with this approach is that one can design much more compact rule sets if multiple rules (resolved by a priority mechanism) can apply to any packet. The paper [EMU] investigates efficient data structures to process basic firewall queries.

The paper [VPR] addresses problems of generating and analyzing rule sets for networks with multiple firewalls and addresses the problem of checking equivalence of rule sets. However, the paper relies on exhaustive analysis of all possible packets, and it is unclear whether the methods scale to large networks. The paper [BRR] describes a tool to configure and analyze rule sets for networks with multiple firewalls. It employs efficient algorithms that scale for large networks, and while the techniques are relevant for checking rule set equivalence, the paper does not explicitly address the equivalence problem.

The paper [MKE] proposes the use of binary decision diagrams to analyze rule sets. These diagrams allow you to query the firewall rule set – but do not easily support the comparison of two rule sets.

References

[ALG] Algorithmic Security Inc., Firewall Analyzer: Make your firewall really safe (Whitepaper), 2006, www.algosec.com.

[AH1] E. Al-Shaer and H. Hamed, Firewall policy advisor for anomaly discovery and rule editing, *Proceedings IEEE/IFIP Integrated Management Conference,* 2003.

[AHB] E. Al-Shaer, H. Hamed, R. Boutaba and M. Hasan, Conflict Classification and Analysis of Distributed Firewall Policies, JSAC 2005.

[BHR] S. Bhatt, B. Horne and P. Rao, The Vantage Firewall Analyzer Prototype, HP Labs Technical Report, 2006.

[BRR] S. Bhatt, S. Rajagopalan, P. Rao. Automatic Management of Network Security Policy, Proceedings *MILCOM,* 2003.

[EMU] D. Eppstein and S. Muthukrishnan, Internet packet filter management and rectangle geometry, *Proceedings ACM SODA*, 2001.

[GLI] M. Gouda and X-Y. Liu, Firewall design: consistency, completeness and compactness, *Proceedings IEEE Intl. Conference on Distributed Computing Systems,* 2004.

[MKE] R. Marmorstein and P. Kearns, An open source solution for testing nat'd and nested iptables firewalls, *Proceedings LISA,* 2005.

[MWZ] A. Mayer, A. Wool and E. Ziskind, Fang: A firewall analysis engine, *Proceedings IEEE Symposium on Security and Privacy*, 2000.

[SOL] Solsoft Inc., www.solsoft.com

[VPR] FACE: A firewall analysis and configuration engine, P. Verma and A. Prakash, *IEEE Symposium on Applications and the Internet,* 2005.

[WOO] A. Wool, Architecting the Lumeta firewall analyzer, *Proceedings 10th USENIX Security Symposium*, 2001002E