End-to-End Network Access Analysis

Sruthi Bandhakavi, Sandeep Bhatt, Cat Okita, Prasad Rao

HP Laboratories
HPL-2008-28R1

Keyword(s):
No keywords available.

Abstract:
Network security administrators cannot always accurately tell which end-to-end accesses are permitted within their network, and which ones are not. The problem is that every access is determined by the configurations of multiple, separately administered, components along a path. Furthermore, configurations evolve over time, and a small change in one configuration file can have widespread impact on the end-to-end accesses. Short of exhaustive testing, which is prohibitively time consuming and impractical, there are no good solutions to analyze end-to-end flows from network configurations. This paper presents a technique to analyze all the end-to-end accesses from the configuration files of network routers and firewalls. The contributions of this paper are to engineer solutions for real network instances that are based on (i) generic templates for network components and (ii) a more general treatment of firewalls, including ways to deal with certain state-dependent filter rules, and (iii) efficient generation of firewall access control rules to meet desired end-to-end flow requirements. Our goal is to help network security engineers and operators quickly determine configuration errors that may cause unexpected access behavior.

External Posting Date: November 21, 2008 [Fulltext] Approved for External Publication
Internal Posting Date: November 21, 2008 [Fulltext]

© Copyright 2008 Hewlett-Packard Development Company, L.P.

End-to-end Network Access Analysis

Sruthi Bandhakavi[*], Sandeep Bhatt, Cat Okita[†], Prasad Rao
Hewlett-Packard Laboratories
5 Vaughn Drive Princeton, NJ 08540

Abstract

Network security administrators cannot always accurately tell which end-to-end accesses are permitted within their network, and which ones are not. The problem is that every access is determined by the configurations of multiple, separately administered, components along its path. Furthermore, configurations are constantly evolving, and a small change in one configuration file can have widespread impact on the end-to-end accesses. Short of exhaustive testing, which is prohibitively time consuming and impractical, there are no good solutions to analyze end-to-end flows from network configurations.

This paper presents a technique to analyze all the end-to-end accesses from the configuration files of network routers, switches and firewalls. Our goal is to help network security engineers and operators quickly determine configuration errors that may cause unexpected behavior such as unwanted accesses or unreachable services. Our technique can be also be used as part of the change management process, to help prevent network misconfiguration.

We build upon the work in [6], which presented an abstract formulation of the problem. The contributions of this paper are to engineer solutions for real network instances that are based on (i) generic templates for network components and (ii) a more general treatment of firewalls, including ways to deal with certain state-dependent filter rules, and (iii) efficient generation of firewall access control rules to meet desired end-to-end flow requirements.

1 Introduction

This paper focuses on the problem of determining all possible end-to-end accesses in a network from the static configurations of network routers and firewalls. We illustrate the problem by describing a typical environment, with three related examples that, extrapolated to enterprise scale networks, are difficult and costly to diagnose and correct.

Figure 1 shows a common network configuration for a managed service provider to provide data center space and management services for enterprise customers. The network is broken into three parts - the customer compartment, an access compartment, which is shared between multiple customers, and the management and monitoring compartment. The network was designed to be highly available, with multiple potential paths through each compartment, depending on the status of the various devices.

The core of the customer installation is a pair of layer 3 switches (`Cust_Switch-1` and `Cust_Switch-2`) in a failover configuration. The switches were connected to the management and monitoring infrastructure of the service provider via a pair of firewalls in an active/passive failover configuration.

[*]Current address: Department of Computer Science, University of Illinois at Urbana-Champaign. This work was done as a summer intern at HP Laboratories.
[†]HP Services Global Delivery ITO

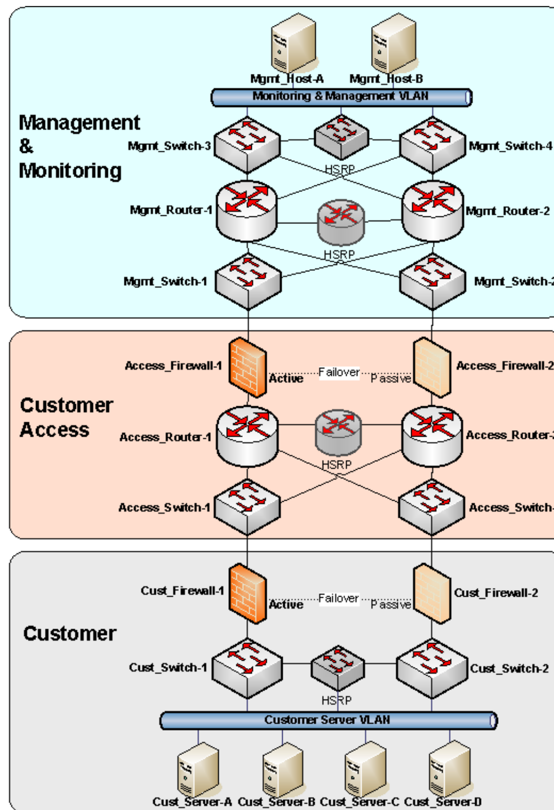

Figure 1: A Single–Site Managed Service Provider Network

Our first issue was straightforward - monitoring traffic from the Management and Monitoring zone did not reach the customer servers. This issue is most often due to missing or misconfigured access control lists (ACLs) or routes.

A check at the Customer firewall, showed that no traffic was being received from the monitoring hosts. When the Access Firewall was examined, it was immediately apparently that there were no ACLs in place to permit traffic to pass from the Management and Monitoring compartment to the customer.

Shortly after the initial monitoring issue had been resolved, the customer requested that their existing site be duplicated at a remote location, to provide redundancy and disaster recovery capabilities.

As shown in Figure 2, the compartments were essentially duplicated between the two sites, Boston and Atlanta. The customer provisioned a primary and backup fiber link between the sites to extend their network and moved half of their existing servers to the new site. The Management and Monitoring VLAN was also extended between the two sites.

Since the Atlanta site was designed to be fully redundant, the expectation was that monitoring traffic for devices in Atlanta would be routed through Atlanta, and monitoring traffic for Boston would be routed through Boston.

While installing the Atlanta site, a number of issues preventing management and monitoring of the new

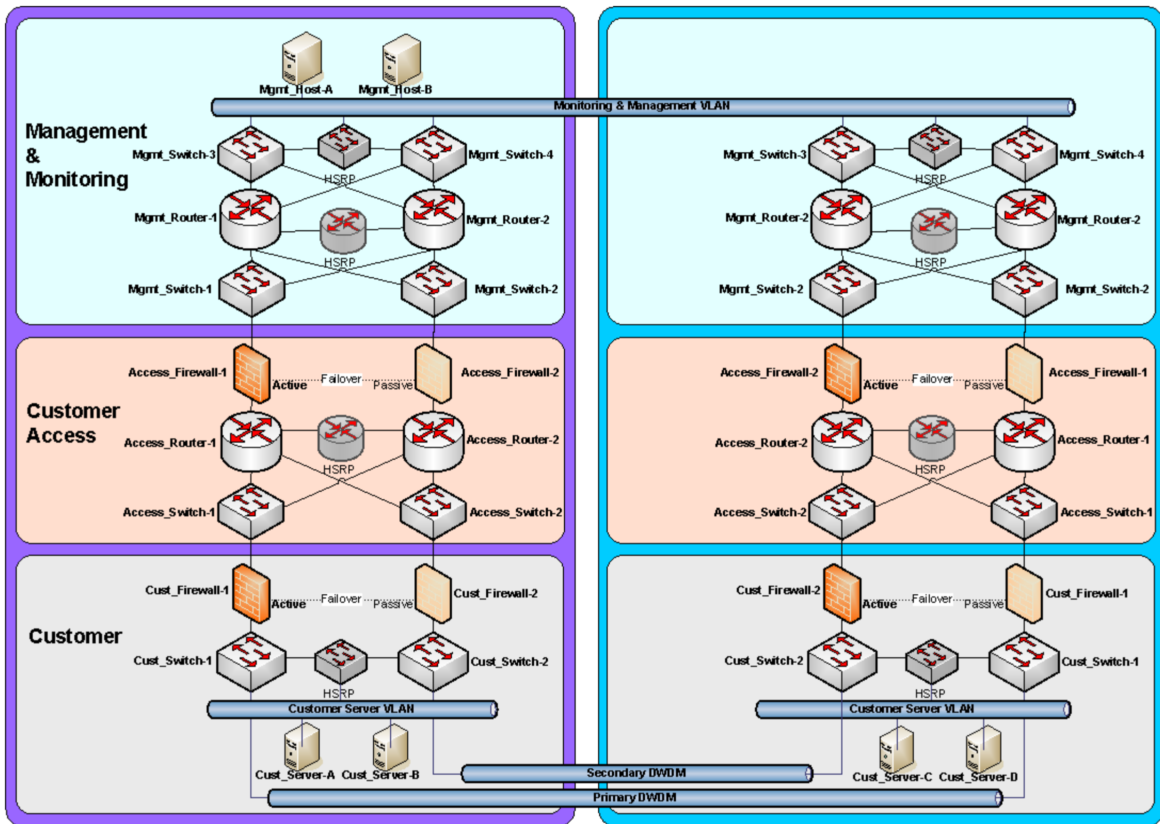

Figure 2: A Dual–Site Managed Service Provider Network, Boston on the left and Atlanta on the right.

servers were discovered. Ultimately, each issue was caused by one or more configuration errors, but the process of identifying and resolving (if possible) each issue provided significant challenges.

Initial tests of monitoring and management for the new site revealed that there was no monitoring connectivity between the Management and Monitoring compartment to the customer servers at the new site.

Initial debugging focused on the Atlanta firewalls, and an immediate issue was discovered - the firewalls did not have a route to the monitoring servers.

Once a route to the Management and Monitoring compartment in Atlanta had been added to the Atlanta customer firewalls, the nature of the problem shifted. As shown in Figure 3, when monitoring connections were attempted from the Management and Monitoring zone to the Atlanta Customer Server VLAN, ICMP monitoring for up/down status was usually successful, but SNMP monitoring and SSH traffic failed.

Since the previous issue had been the result of a firewall misconfiguration, and ICMP connections were successful, the debugging process continued to focus on the Atlanta customer firewalls.

It soon became evident that the monitoring traffic destined for the Atlanta customer servers was not arriving via the Atlanta firewalls. A check of the Boston firewalls determined that the monitoring traffic destined for the Atlanta servers was being routed incorrectly, via Boston, and arriving in Atlanta via the customers' internal network.

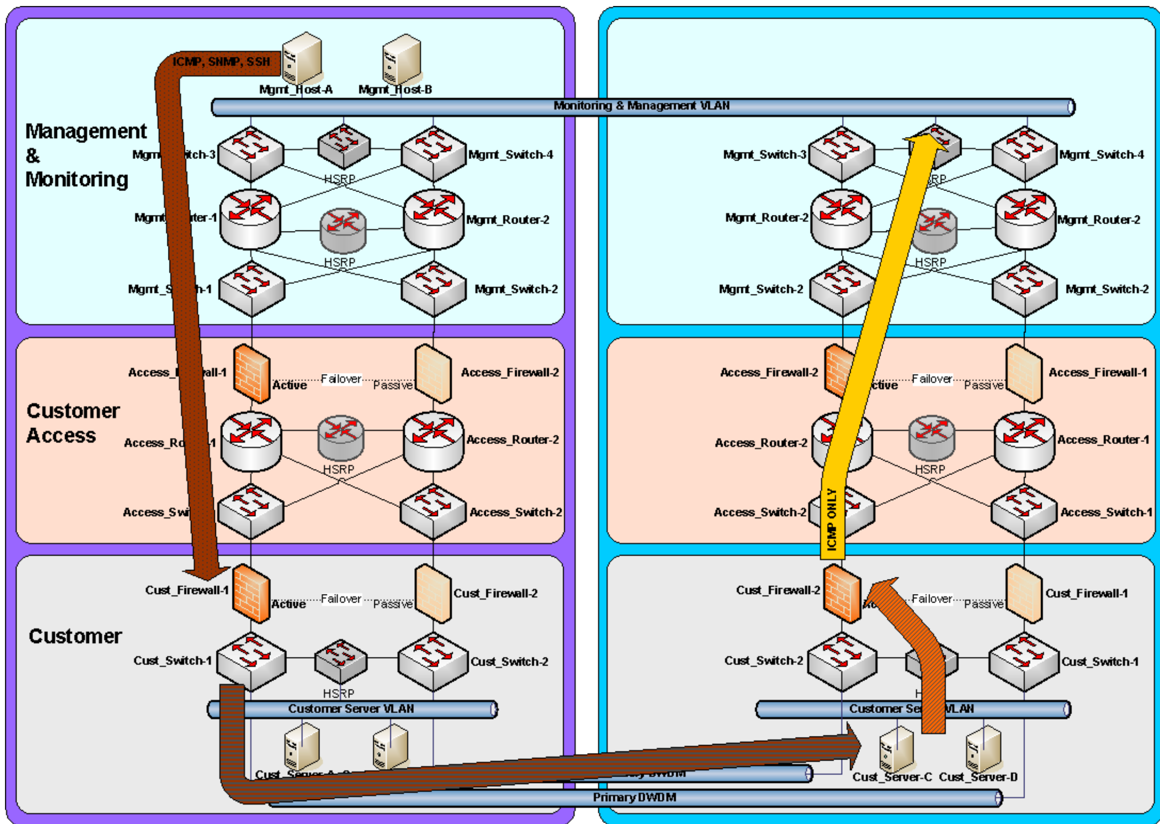

Figure 3: Misconfigured routes and firewalls denied monitoring services

Like most modern firewalls, the firewalls used the *keep state* feature; using this feature the firewalls track TCP and UDP connections that have been started, and drop any traffic for which they do not have a starting record. ICMP is not considered to be stateful. Thus, while the Boston firewalls recorded the start of inbound connections from the Monitoring host to the Server VLAN in Atlanta (that were incorrectly routed through Boston), returning traffic was correctly sent via the Atlanta firewalls, which did not have a record of the inbound traffic, resulting in the traffic being dropped. On the other hand, since ICMP is not stateful, returning ICMP was permitted to exit the Atlanta firewalls, and ultimately returned successfully to the Management and Monitoring compartment.

Due to time pressures, the initial solution to the monitoring issue was to modify the customer routing, to return all monitoring traffic through Boston. This created a single point of failure, as a failure in Boston would result in the loss of management and monitoring of the Atlanta servers.

Unfortunately, while the monitoring issues appeared to have been resolved, every time the layer-3 switches in Atlanta were rebooted, monitoring for all hosts in Atlanta would fail again. The problem was that on the Cisco router the floating static routes (which generally have lower precedence than the dynamic routes) come up first during a reboot and get precedence over the dynamic routes [3]. This is a classic example of a latent failure which can be caught by configuration analysis.

In all examples above, the end-to-end failures resulted from unintended interactions between the misconfigurations of the router and the firewall access control rules. In a large enterprise network that is managed by geographically distributed teams of operations engineers, problems such as these can take anything from hours to weeks to diagnose and fix. Moreover, with hundreds or thousands of routers and firewalls, the potential interactions between configurations are numerous and the cost of manually determining all end-to-end flows is prohibitive.

The techniques developed in this paper will help discover issues such as those raised in the examples above. For example, given the set of router and firewall configurations, our techniques would discover the possibility of asymmetric routes interacting with firewall rules to block SNMP and SSH connections between the management and monitoring compartment and the Atlanta customer servers. A tool based on our techniques, capable of statically analyzing network configurations, would be useful both in the design of network configurations and in troubleshooting networks when they misbehave.

1.1 Problem Statement

This paper investigates the problem of determining all end-to-end conectivities in a network, given the set of all router and firewall configuration files in the network. The configuration data includes firewall access control rules, router connectivity and route advertisement policies.

While the behavior of a network at any given point in time depends on the state of the routing tables, the behavior can change as the dynamic routing tables change. It is quite possible, therefore, for a network to behave correctly at one instant, but behave in an unintended and incorrect manner at a later point in time.

In this paper we are not concerned with the analysis of a network at any given instant in time. We do not examine the state of routing tables. Instead, we consider the set of all potential paths from a source to a destination, over all possible states of the network routing tables. We can analyze all potential source-destination paths by examining the static route advertisement policies and the connectivity of network elements.

Now, certain packet types (specified by port and protocol) between the source and destination may be blocked by firewall rules along some (or all) of these paths. For any packet, there are three outcomes: (i) all paths from source to destination are blocked, (ii) no path is blocked, or (iii) some paths are blocked while others are not.

1.1.1 End-to-end Requirements

An end-to-end requirement specifies, for a given source-destination pair which packet types must flow through and which must be denied. Such a requirement implies that either every possible path must allow the packet to go through, or else no path must allow the packet to go through. In other words, satisfying an end-to-end requirement means that the third option (some paths permit the packet while others block it) must be ruled out by the static configuration settings. This type of deterministic behavior in networks is critical to the long term security and stability of the environment.

As we saw in the previous section, the routing policies may, perhaps unexpectedly, allow asymmetric routes — the route from source to destination may be different from the path back from the destination to the source. Asymmetric routes can be hard to detect manually, and even harder to debug. If we can examine all the possible round-trip paths between a source and destination, we will implicitly be finding all the possible routing asymmetries, and can make the administrator aware of them.

Formally, an end-to-end access requirement is represented as

$$< source, destination, service, permission, priority >$$

where *source* and *destination* correspond to sets of IP_addresses, *service* contains the source and destination ports, and protocol, *permission* is either "allow" or "deny," and *priority* is a unique rank assigned to the requirement. A set of requirements is thus rank-ordered.

The main problem we address in this paper is: given a set of end-to-end access requirements, and a set of router and firewall configuration settings, do the configurations implement the end-to-end access requirements? For requirements that are not satisfied, find paths that violate the requirement.

Furthermore, if any requirements are not satisfied, give an algorithm to compute new firewall rules, if they exist, to satisfy all the end-to-end access requirements. In this paper we discuss how to generate new rules. Our intent is not to automatically manage firewall rule sets, which involves subtle and complex issues, but rather to propose new rules to be vetted by an expert. We do not consider the problem of automatically reconfiguring local routing policies to satisfy end-to-end requirements.

1.2 Solution Approach

Our approach to analyzing end-to-end network reachability is to determine all possible end-to-end accesses from the static configurations. This is similar to the approach taken in [6] which presents an abstract framework to address the problem. As mentioned in [6], the dynamic behavior of a network depends on factors beyond static configurations, for example, the actual route traversed depends on the dynamic state of the routing tables, and is not determined simply by the static configuration settings. However, static analysis does accurately capture reachability properties that the network is required to uphold in every possible dynamic state of the routing tables.

We define generic model templates for router and firewall configurations. For a given network, an instance of the model is created for each device, and the template is populated with data drawn from the device configuration files. From the configuration parameters in the model instances we construct route graphs for the network. These graphs encode all possible paths between any two end points, taking into account routing policies and firewall rule sets. Once the route graphs are constructed, we determine the set of all end-to-end accesses, and check these against the set of end-to-end requirements.

In summary, our solution to check the end-to-end access requirements proceeds in the following stages:

1. **Model Instantiation.** For each router and firewall, create a model instance by populating the model template with parameters from the configuration files.

2. **Route Calculation.** From the model instances, create a route advertisement graph. Each node of the graph represents a routing process. Edges in this graph are used to propagate the route advertisement entries between routers. The routes are consolidated into one Routing Information Base in each router model instance.

3. **Route Analysis.** Create Route graphs, one per destination. The nodes of each graph are routers, firewalls and subnets, and each destination route graph captures all possible paths to the destination from all sources. Use the route graphs to calculate all the end-to-end connectivities.

4. **End-to-end validation.** Check the set of all end-to-end accesses against the set of end-to-end requirements to find any violations, and suggest fixes wherever possible.

1.3 Outline of this paper

The remainder of this paper is organized as follows. Section 2 gives an overview of the generic models for routers and firewalls. Section 3 describes how the configuration files are parsed and the results used to populate the model instances. Section 4 defines the route advertisement graph, the calculation of the routing information base for each router, and defines the route graphs which are used in Section 5 to calculate all the possible end-to-end accesses. Section 6 describes previous related work, and Section 7 concludes with a list of issues for further exploration.

2 Defining and Populating Models

This section presents our models for routers and firewalls. The models are generic and intended to support different vendor products and versions. Furthermore, each configuration model represents the minimum information required to generate the required reachability information; thus, we only define attributes that capture the routing policies and access control rules that concern us. In the next section we show how to derive the information required by this generic model from real configurations.

For ease of exposition, our model separates routing and firewall functions; router and switch ACLs are considered as a separate "firewall" associated with the router or switch in question.

2.1 Router Model

The model shown in 4 allows us to describe, and later capture the vendor-independent information about the routing device required for our purposes. In our model, a router or layer 3 switch[1] consists of an identifier associated with a set of routing methods, which are then combined with the associated physical and virtual interfaces.

Any given router will typically have a combination of directly connected networks, static routes and dynamically learned routes, such as routes learned via dynamic routing protocols such as RIP, OSPF or BGP.

We present the salient features of the model to give a general idea of the structure of a router. For clarity, we defer some details to SectionA.1 which describes, with an example, how the model is populated as well.

2.1.1 Routes and Route Models

We describe routes as a 3-tuple «destination network, next hop, weight», where destination network is the desired destination, next hop is the address to which packets must be sent in order to reach that network, and weight is a number which can be used to indicate route preference. This definition is common to all routing methods.

2.1.2 Directly Connected Routing and Route Models

Each router is a member of multiple networks to which its interfaces are connected. Each of these networks is used to populate the field labeled "directly connected routes."

[1]For simplicity, we will use the term 'router' from this point onwards to refer to either a router or a layer 3 switch.

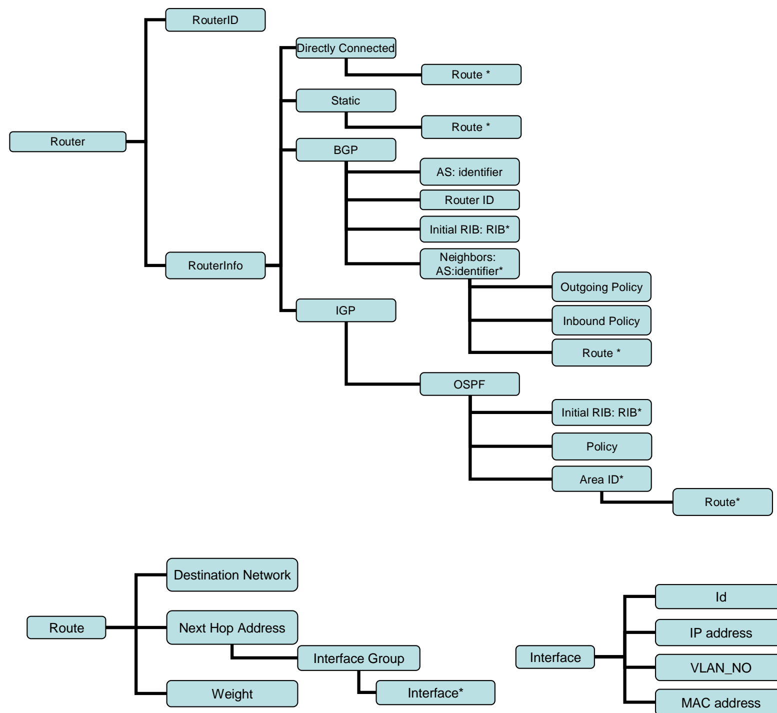

Figure 4: Router Model

2.1.3 Static Routing and Route Models

Static routes are fixed paths defined on the router, rather than learned via a routing protocol. Since fixed routes are fragile in the face of network failure, static routes are typically used as a route of last resort, or to point to a shared virtual address in HSRP and VRRP configurations. The static route model simply consists of a set of fixed routing rules which specify the next hop to which a packet will be routed.

2.1.4 Dynamic Routing and Route Models

Dynamic (or adaptive) routing protocols update the available network paths inside of an autonomous system (AS, loosely described as a group of routers sharing the same administrative policies) and between ASes in realtime (or near real time) based on changes that occur in the routing environment.

Routing protocols used inside of an AS are described as Interior gateway protocols (IGP). Functionally there is only one Inter-AS protocol in use – BGP.

Briefly, there are three general types of algorithm used in dynamic routing:

1. Distance Vector (RIP, IGRP, EIGRP)
2. Link State
3. Path Vector

In the Appendix we describe models for intra-AS routing using OSPF and inter-AS routing with BGP.

2.1.5 Access Control List Models

A router can also specify access-control lists to control incoming and outgoing packets, represented by Router.Incoming Policy and `Router.Outgoing Policy`. In this paper we assume, for sake of simplicity, that all filtering is done at firewalls. For routers with filters we will represent it as a combination of a router and a set of firewalls, one connected to each interface.

2.2 Firewall Model

The model shown in Figure 5 allows us to describe and later capture the vendor-independent information about firewalls and ACLs required by our algorithms. In our model, a firewall consists of a set of interfaces, to which network address translations and policies to permit or deny access may be applied. We describe the salient features here.

2.2.1 Interfaces

The firewall interfaces are a set of all of the interface ids and IP addresses of the subnets physically connected to various interfaces of the firewall.

2.3 Order of Operations

While the set of operations (route, translate, apply policy) performed by any type of firewall is fairly consistent, the order in which these operations are performed varies. The firewall which we will be using as an example, OpenBSD pf, uses «Translate, Apply Policy, Route». Cisco IOS ACLs use «Apply Policy, Route, Translate» for traffic coming from the "inside" interface to the "outside" – but «Apply Policy, Translate, Route» for traffic coming from the "outside" interface to the "inside".

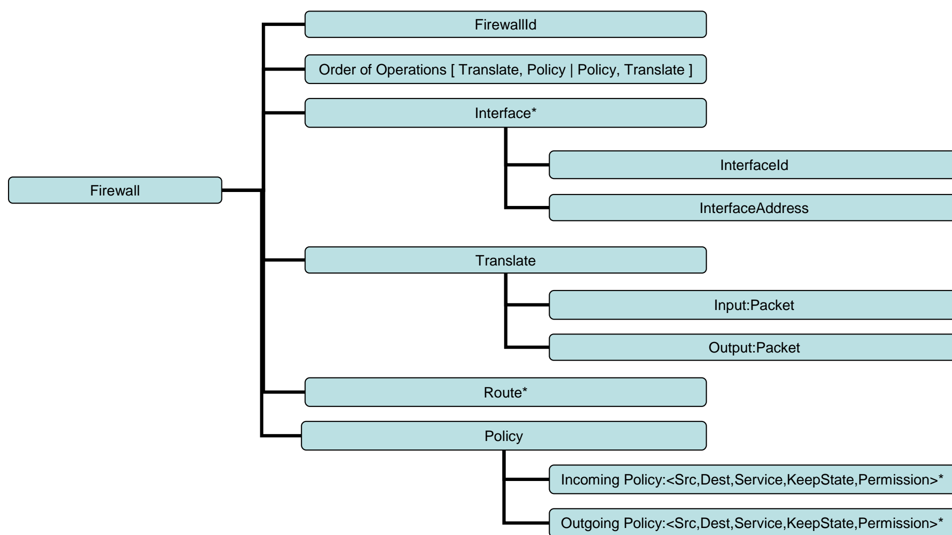

Figure 5: Firewall Model

2.4 Routes and RIB

This is the set of static routes and directly connected routes known to the firewall. For firewalls with dynamic routing, we represent the firewall as a combination of a firewall and a set of routers, one connected to each interface.

2.4.1 Policy

Policy is what determines if a packet will be accepted or denied, and depending on the type of firewall or access list, may be applied to packets as they are incoming, outgoing, or both.

1. IncomingPolicy consists of a set of all of the policies that are applied to traffic inbound through a particular interface. Each entry consists of the 4-tuple <Src,Dest,Port,Protocol>, and a boolean `Keep State` attribute (which is true if this rule was created as a result of an implicit "keep state" rule and it is false if the packet filter rule does not have keep state).

2. OutgoingPolicy consists of a set of all of the policies that are applied to traffic outgoing through a particular interface. Each entry consists of the 4-tuple <Src,Dest,Port,Protocol>, and a boolean `Keep State` attribute (which is true if this rule was created as a result of an implicit "keep state" rule and it is false if the packet filter rule does not have keep state).

Some types of firewall have implicit rules such as "packets are allowed to travel from higher security interfaces to lower security interfaces, but not vice versa", which means that the policy may need to be interpolated.

2.4.2 Translation

Translation is an general description for mechanisms that modify the source, destination or ports described by firewall policies. Translation includes network address translation (NAT), port address translation (PAT) and redirection (RDR), which are briefly described here.

1. Network Address Translation consists of a set of translation rules which are applied as packets transit a given interface. NAT works as a function which takes certain source or destination addresses and converts them into other addresses.

2. Port Address Translation consists of a set of translation rules which take certain source or destination ports, and convert them into other ports.

3. Redirect (RDR) consists of a set of translation rules for destination addresses and/or ports.

3 From Configurations to Models

The models in the previous section are represented using java beans as classes with getter and setter methods. These models have:

1. A java constructor to create model nodes by invoking *new* with the classname.

2. Setter methods to assign values to fields in model nodes – `object.setField(value)`. Values are either model nodes or scalar items such as integers and strings.

The configuration of a device is a language that is subject to syntactic and semantic rules, contained in a grammar.

We use this grammar to parse the configuration file into a parse tree. This parse tree maps rules of the grammar to the contents of the configuration file. The leaves of the parse tree together are the contents of the configuration file. The interior nodes of the parse tree are instantiated versions of grammar rules.

We populate the models by traversing the parse tree in preorder. On encountering an interior node we execute the actions specified in the body of the rules. At the completion of this traversal we will have a completely populated model of the device, provided we started off with a valid configuration file, and a correct grammar for the device.

3.1 A Brief Example

We represent a routing rule in Figure 4 using the java class

```
class NextHopInfo{
IpAddress nextHopInfo;
Vector<Inferface> interfaceGroup;
}


class Route{
IpAddress DestinationNetwork;
NextHopInfo NextHop;
float weight;
}
```

We have to populate instances of the above model fragment using the grammar rules[2] :

$RouteSpec \rightarrow NetworkSpec\ NextHopInfo\ Weight;$
$NextHopInfo \rightarrow NextHopAddress\ InterfaceGroup;$

We annotate the rules above with actions to populate fields in the models using setter methods.

```
RouteSpec →
$$ = new Route();
NetworkSpec
{ $$.setNetworkSpec($1); }
NextHopInfo
{ $$.setNextHopInfo($2); }
Weight
{ $$.setWeight($3); }
;
NextHopInfo →
{ $$ = new NextHopInfo(); }
NextHopAddress
{ $$.setNextHopAddress($1); }
InterfaceGroup;
{ $$.setInterfaceGroup($2); }
```

[2]In Appendix A.1 we show the population of models intuitively without resorting to lex and yacc conventions. In this section, however, we use lex/yacc conventions for convenience.

The action `new Route();` creates a `Route` model instance, when the grammar rule for $NetworkSpec$ is triggered. Then the `destinationNetwork`, `nextHop` and the `weight` fields are filled by calling the setter methods `setNetworkSpec`, `setNextHopInfo` and `setWeight` on this instance. The arguments for these methods are obtained by the recursive traversal of the parse tree.

4 Computing Route Graphs from Populated Models

In this section we first show how to construct the *route advertisement graph* (RAG) from a set of populated router and firewall models. We use the RAG to calculate the consolidated route information base (RIB) for each node in the RAG. Using the RIBs we then construct route graphs from which the end-to-end accesses will be calculated.

4.1 Route Advertisement Graph(RAG)

We first define the route advertisement graph. The RAG, $G(V, E)$ contains a node for each routing process ($R_i.BGP$, $R_i.OSPF$) and for the static route set $R_i.Static$ in a router. The edge set E consists of all edges $x.p_x \rightarrow y.p_y$ if x's routing process p_x advertises to y's routing process p_y.

We can determine whether a routing process advertises routes to another based on the populated models of the two processes. For instance, one of the conditions under which the BGP process of a router x advertises routes to the BGP process of another router y (using names taken from Section 2):

- if y.BGP $\in$ x.BGP.neighbors and
- (x.BGP.AS $\neq$ y.BGP.AS) and
- $\exists I \in x.Interfaces, I' \in y.Interfaces : I.VLAN_NO = I'.VLAN_NO$.

4.2 Propagating the Routing Information Base

Once the RAG has been created, we use it to compute the set of routes available to each host. As described next, this involves propagating the information in each RIB throughout the network, and updating every RIB as it receives new route information. As this process is iterated, each RIB eventually converges to a fixed point; this final state determines all the network addresses to which the router can route traffic.

A standalone router can calculate its reachability information by looking at the local RIBs and the RIBs of all the routing processes. If there is a directed edge from routing process x.proto1 to y.proto2, where x and y are routers, then in the absence of any access control policies, [3] x.proto1 sends the list of destinations it can reach to y.proto2. This means that y.proto2 inherits routes to all the destinations reachable from x.proto1.

The outgoing policy of x.proto1 and the incoming policy of y.proto2 determine exactly which routes of x.proto1 can be propagated to y.proto2. Therefore, using the static information from the router's configuration, we can update the set of destinations that can be reached from a particular router interface or through which the router can send a packet to a particular destination.

We use the algorithm in Figure 1 to compute the fixed point values of the RIBs. Upon reaching the fixed point, each router's consolidated RIB (R.RIB)is calculated as the union of RIBs of all the individual

[3]The reader should note that routing policy restrictions are clearly separate from the firewall access controls. The former controls routing advertisments, whereas the latter are used to control network access.

Algorithm 1 Route Propagation Algorithm

```
1:  procedure RIBPropogate(RIBGraph)
2:      repeat
3:          for all (R1.proto1, R2.proto2) ∈ RIBGraph.Edge do
4:              for all routes ∈ R1.proto1.RIB do
5:                  if Policy allows advertising route then
6:                      R2.proto2.RIB.add(route)
7:                  end if
8:              end for
9:          end for
10:     until RIB FixedPoint is reached
11: end procedure
```

routing processes and the local RIB which also contains the static routes. At this point, each consolidated RIB contains the following information:

- All the destinations reachable from the router, and
- For each reachable destination IP in the RIB, the list of next-hop interfaces.

4.3 Route Graphs

From the RIBs computed as the fixed point of the propagation algorithm, we next define a set of route graphs. These route graphs will be used in the next section to verify whether or not the network configurations satisfy the end-to-end access requirements.

We start with a set of destination graphs, G_d, one for each destination node d in the network. The destination graph G_d contains a node for each router, firewall, subnet and VLAN. There is a directed edge from node x to node y if a packet destined for node d can possibly traverse the link from x to y.

Next, we shrink G_d as follows:

1. First, remove all firewall nodes from G_d. This splits the graph into a number of disconnected subgraphs.
2. Within each subgraph, identify every strongly connected component and replace it with a single "supernode," while maintaining the external incoming or outgoing edges for the component.
3. Finally, insert the firewalls back in, making connections to the appropriate nodes/supernodes. We call this residual graph the route graph R_d.

Note that each route graph R_d contains within it all the possible paths along which a packet destined for d can travel from any source.

Figure 6 shows route graphs for the two nodes `Cust_Server-D` and `Mgmt_Host-A` from the example of Section 1. The solid edges belong to the route graph for `Cust_Server-D`, and the dotted edges belong to the route graph for `Mgmt_Host-A`.

Although global enterprises have large networks, with tens of thousands of elements, the number of firewalls is typically in the low hundreds for the largest networks and in the low tens for typical networks. The route graphs defined above are considerably smaller than the original networks; this flexibility allows us the flexibility to search over all possible source-destination paths in the next section.

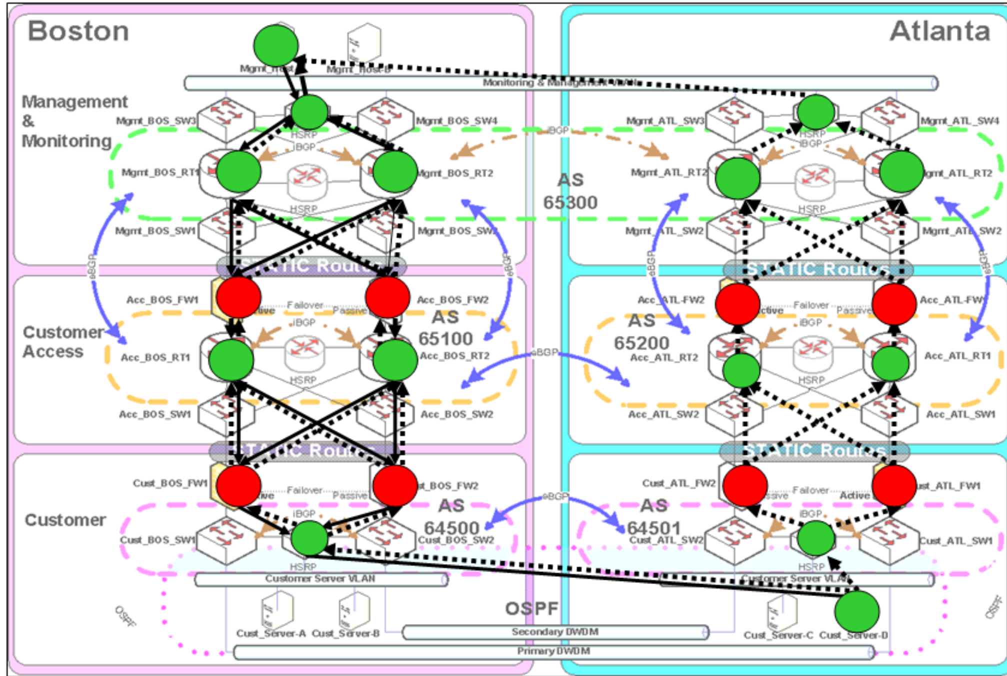

Figure 6: Illustration of route graphs

5 Reachability

Given the route graphs R_d for each destination, we next compute all the end-to-end accesses in the network that are consistent with the router setup, route filters, and firewall access control policies.

A high-level overview of the algorithm is presented in Figure 2. We omit details that, while important, would detract from clarity.

The algorithm takes as input a set of directed route graphs, one graph for each destination. Each firewall node is labeled with an associated rule set (subset of the original rule set containing only rules that match the destination node d).

For every source and destination pair (s, d) the algorithm classifies all services $(port, protocol)$) according to whether (a) every path from s to d is open (not blocked by a firewall), (b) every path from s to d is blocked, or if (c) some s to d paths are open while others are blocked.

One detail that we have ignored is that while nodes in our graph correspond to aggregates such as VLANs and subnets, firewall rules may apply at a finer level of granularity, perhaps even to individual IP addresses. While we do not go into details in this paper, such issues can be handled in a straightforward manner; for

Algorithm 2 Classifying services between source and destination

1. Input: The route graph R_d and a source node s.
2. Output: Label each service $(port, protocol)$ according to whether all s to d paths are (i) blocked, (ii) open, or (iii) some paths are blocked while others are open.
3. Maintain two sets *Open* and *Blocked* at node d. The set *Open* contains services for which some path from s was open, while *Blocked* contains services for which some path was blocked. Initially both sets are empty.
4. Begin a search at the source s that exhaustively searches every path from s to d. We maintain a list L (initially empty) of the services that are blocked along the path P (initially containing the source s) currently being explored.
   (a) When the path encounters a firewall, add the list of services that are blocked for d to list L.
   (b) When the path reaches d, insert the elements of L into the list *Blocked* maintained at node d. Similarly, the services not in L are inserted into the list *Open*.
5. When all s-to-d paths are exhausted, label each service according to whether it appears in only one of the lists *Open* or *Blocked*, or in both lists.

---

example, by refining each node into smaller subnodes depending on applicable rules across all firewalls.

Note that the running time is proportional to the number of source–destination paths, plus the time for bookkeeping operations to maintain the sets. There are several improvements to the algorithm that we have omitted in order to keep the presentation clear and simple.

Although, in the worst case, the number of paths can be exponential in the size of the route graph, usually the route graph is substantially smaller than the original graph. Furthermore, production networks are structured so that the number of firewalls along any path, as well as the number of source–destination paths is severely limited. In practice, we expect the algorithm will scale well for large enterprise networks.

We note that there are other ways to solve the problem that do not suffer from this exponential worst case behavior. For example, by examining services (port, protocol pairs) one at a time, each iteration takes time linear in the size of the route graph, and the worst case time is proportional to the product of the graph size (number of nodes and edges), and the number services (ports, protocols pairs). In practice, the last two factors in the product make the running time on typical networks much worse than the algorithm given above.

Finally, having computed all the end-to-end accesses, it is a straightforward matter to verify if these are compliant with respect to a given set of end-to-end requirements.

5.1 Round trip flows and Keep State Rules

The above discussion considered only one-way flows and stateless firewalls. For round-trip flows (tcp, http, ftp, etc.) the situation is complicated by the presence of stateful firewall rules, in particular "keep state" rules. In conjunction with asymmetric routes that are created unintentionally as a result of a misconfiguration, stateful rules make network problems difficult to debug.

Our approach to analyzing round-trip flows proceeds as follows: start with the route graphs R_d and R_s,

and merge/concatenate the two graphs by identifying node d in the two graphs. In the concatenated graph, a path from s to d can be concatenated with a return path from d to s. We can use the previous algorithm to find all the round-trip flows from s to d and back.

Now, some of the firewalls may contain "keep state" rules which work as follows. Suppose that, on the return path from d to s, a firewall is encountered and the rule triggered permits the return packet, but is marked keep-state. Then, the packet is allowed through only if the forward path from s to d transited through the same firewall; if not, the return packet is dropped. Generalizing this observation, it is easy to see that in order to satisfy an "allow" end-to-end requirement on a round-trip service, *every* forward path must pass through a firewall with a keep-state rule if even one return path passes through that firewall.

To account for the keep state rules, we thus modify our search procedure accordingly. Specifically, while exploring a round-trip path, if a "keep state" firewall rule is encountered on the return path, then we check to see if that firewall was on the forward portion (from s to d) of the current path. If not, the service is blocked along that path. We leave the details of the algorithm to the interested reader.

5.2 Reconfiguring firewall rules

Suppose that we are given a set of end-to-end access requirements of the form

$$(s, d, port, protocol, permission),$$

where *permission* is either *allow* or *deny*, and our reachability analysis reveals that some of these requirements are not always satisfied. In such cases, we would like to identify which devices require configuration changes, and what changes need to be implemented.

A naive approach is to compute the fixes as we run the search procedure. In particular, for an *allow* requirement, we need to ensure that all firewalls along every $s - d$ path will permit the packet through. During the search, we simply add the appropriate rule along every firewall encountered. Similarly, for a *deny* requirement, we need to ensure that every $s - d$ path blocks the packet; for example, we could choose to block firewalls that form an $s - d$ cut set computed by a breadth-first search of the paths from s to d, and insert deny rules for the packet appropriately.

In general, we do not recommend that changes to firewall rule sets be made automatically. Rather, the rules generated above could be a starting point for an administrator responsible for refining the rule set. One weakness of the naive approach is that by adding rules for indivdual packets, we could create very large, and therefore perhaps inefficient, rule sets. While there are promising approaches to aggregating rule sets, these are beyond the scope of this paper.

6 Related Work

The paper by Xie et.al. [6] presented an abstract framework to study the static reachability problem. The framework is based on the use of dynamic programming to compute all possible accesses, aggregated over all possible states. They consider rules that are limited to filtering on outgoing destinations only, and not on packet sources. Moreover, their algorithm does not address the issue of composing the effects of firewall rule sets along source-destination paths.

We have built on their ideas to engineer solutions for real network instances by developing (i) generic models for network components (ii) a more general treatment of firewalls, including ways to deal with state-dependent filter rules, and (iii) efficient generation of firewall access control rules to meet desired end-to-end access requirements.

Guttman and Herzog [2] present a formal framework for analyzing security properties of networks from device configurations. They study properties beyond end-to-end access; however, their treatment is limited to a single snapshot of the network state — they pull in data from the routing tables, but do not consider policies on route advertisement. The paper acknowledges the difficulty of gathering a consistent snapshot of a large network, as well as the limitations of studying properties of single snapshots.

Several papers have studied the problem of verifying end-to-end reachability in networks with distributed firewalls [1, 4, 5]. These papers do not, however, consider the effect of routing policies on end-to-end access. The algorithm given in [1] for computing end-to-end accesses is similar in spirit to the algorithm in Section 5.

7 Next Steps

Our work reveals a number of avenues for further research. One of the parameters that we have not considered are the weights assigned to links in the configuration of routing policies. For example, these weights can be used to estimate the likelihood of particular end-to-end paths, and to distinguish primary paths from secondary, backup paths.

One can also expand the notion of end-to-end access requirements beyond a simple boolean choice, to include, for example, path contraints. Since we explicitly traverse all source-destination paths, we could include more general path properties in our policy. In the example of Section 1, a reasonable end-to-end policy might be that primary paths from the management and monitoring nodes to the customer servers must not traverse the customer-provisioned link. An understanding of the trade-offs between the expressiveness of policy languages versus the complexity of checking compliance against policies is needed to achieve the correct balance in this trade-off.

Along a different line, further work is needed to understand how complete or extensible the generic router and firewall models are in capturing the relevant parameters for multiple vendor products.

References

[1] Elisha Ziskind Alain Mayer, Avishai Wool. Offline Firewall Analysis. *International Journal of Information Security*, Volume 5(3):125–144, July 2006.

[2] J. D. Guttman and A. L. Herzog. Rigorous Automated Network Security Management. *International Journal of Information Security*, 4(1–2):24–98, February 2005.

[3] IBM. http://www.redbooks.ibm.com/pubs/pdfs/redbooks/gg243376.pdf.

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

[5] Pavan Verma and Atul Prakash. FACE: A firewall analysis and configuration engine. In *SAINT*, pages 74–81. IEEE Computer Society, 2005.

[6] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. On static reachability analysis of IP networks. In *INFOCOM*, pages 2170–2183. IEEE, 2005.

A Appendix: Converting configurations into models

A.1 Configuration Grammars

As explained in Section 3 we use generic grammars for routers and firewalls as a starting point in the process of populating model instances. Here we present the generic grammars for describing router and firewall configurations.

Figure 7 is the generic router grammar, while Figure 8 is the generic firewall grammar. Each rule in a grammar has a left hand side and a right hand side separated by a $\rightarrow$. Symbols in the `typewriter` font appear literally in the configuration file; we call these symbols keywords. Symbols that appear on the left hand side of a grammar are *nonterminals*. Others such as *number* are terminals. Throughout this section, `this style of text` is used to indicate a fragment of the actual configuration.

Figure 8 presents the generic firewall grammar.

A.2 Populating Model Instances

To capture a particular kind of router or firewall, we first construct a specific grammar for the device. Then we will use a tool such as yacc or ANTLR to create an executable parser from this grammar. Figure 9 shows the stages of our process. We start by tailoring the generic grammar to create device grammars for specific device types; next a parser goes over the device configuration file, and applies device grammar rules appropriately to produce fragments of the populated model instance. Rather than describe the details of this process, we present the approach, as applied to Cisco IOS files, in Figure 9.

1. *router* → *RouterID RouterInfo*
2. *routerID* → ***string***
3. *routerInfo* → *connectedRoutingInfo staticRoutingInfo IGPRoutingInfo bgpRoutingInfo*
4. *connectedRoutingInfo* → *routeSpec** ***connected***
5. *routeSpec* → *networkSpec nextHopInfo weight*
6. *networkSpec* → *networkAddress wildcard-mask*
7. *nextHopInfo* → *nextHopAdddress interface-group*
8. *nextHopAddress* → ***IP address***
9. *interface-group* → *interface**
10. *interface* → *interfaceID*
11. *interfaceID* → ***string***
12. *staticRoutingInfo* → *routeSpec** ***static***
13. *IGPRoutingInfo* → *[ospfRoutingInfo | IS-IS | ...]*
14. *ospfRoutingInfo* → *ospfInitialRib*
15. *ospfInitialRib* → *ospfRibEntry**
16. *ospfRibEntry* → *ospf-areas*
17. *ospf-areas* → *ospf-area**
18. *ospf-area* → *routeSpec* ospf-areaID*
19. *bgpRoutingInfo* → *bgpInitialRib bgpAS-ID*
20. *bgpAS-ID* → *ASnumber bgpRouterID bgpRouterDescription*
21. *bgpInitialRib* → *bgpRibEntry**
22. *bgpRibEntry* → *bgpNeighbors*
23. *bgpNeighbors* → *bgpNeighbor**
24. *bgpNeighbor* → *routeSpec* bgpNeighborAS-ID*
25. *bgpNeighborAS-ID* → *bgpAS-ID*

Figure 7: Generic Router Grammar

1. *firewall* → *firewallID firewallInfo*
2. *firewallID* → **string**
3. *firewallInfo* → *operationInfo routeInfo translationInfo policyInfo*
4. *operationInfo* → **(translate,policy | policy,translate)**
5. *routeInfo* → *connectedRoutingInfo staticRoutingInfo*
6. *connectedRoutingInfo* → *routeSpec** **connected**
7. *routeSpec* → *networkSpec nextHopInfo weight*
8. *networkSpec* → *networkAddress wildcard-mask*
9. *nextHopInfo* → *nextHopAdddress interface-group*
10. *interface-group* → *interface**
11. *interface* → *interfaceID*
12. *interfaceID* → **string**
13. *staticRoutingInfo* → *routeSpec** **static**
14. *translationInfo* → *translationType translateFrom translateTo*
15. *translationType* → **(nat | rdr | ...)**
16. *translateFrom* → *ruleObject*
17. *translateTo* → *ruleObject*
18. *policyInfo* → *permission direction sourceInfo destinationInfo*
19. *permission* → **(permit | deny)**
20. *sourceInfo* → *ruleObject**
21. *destinationInfo* → *ruleObject**
22. *ruleObject* → *networkSpec portSpec protoSpec*
23. *portSpect* → *port-range*
24. *port-range* → *ports*
25. *ports* → *port**
26. *port* → **number**
27. *protoSpec* → **(tcp | udp | ip | number)**

Figure 8: Generic Firewall Grammar

Generic Grammar	**Device Specific Grammar**	**Configuration File**	**Value**
router → *RouterID RouterInfo*			
routerID → ***string***	*routerID* → ***hostname*** *string*	`hostname SITEA_SW0`	*routerID* → `SITEA_SW0`
routerInfo → *connectedRoutingInfo staticRoutingInfo IGPRoutingInfo bgpRoutingInfo* *connectedRoutingInfo* → *routeSpec** ***connected***	`connected`		`connected`
routeSpec → *networkSpec nextHopInfo weight*			
networkSpec → *networkAddress wildcard-mask*	*networkSpec* → `ip address` IPaddress wildcard-mask	`ip address 192.168.200.2 255.255.255.0`	*networkSpec* → `192.168.200.0/24`
nextHopInfo → *nextHopAdddress interface-group*			
nextHopAddress → IPaddress	*nextHopAddress* → ***ip address*** *IPaddress*	`ip address 192.168.200.2 255.255.255.0`	*nextHopAddress* → ***192.168.200.2***
interface-group → *interface** *interface* → *interfaceID* *interfaceID* → ***string***	*interfaceID* → ***interface*** *string*	`interface Vlan1300`	*interfaceID* → ***Vlan1300***
			connectedRoutingInfo → `192.168.200.0/24 192.168.200.2 Vlan1300 connected`

Figure 9: Example of progression from generic grammar to Cisco IOS-specific grammar