Inferring Higher Level Policies from Firewall Rules

Alok Tongaonkar, Niranjan Inamdar, and R. Sekar – Stony Brook University

ABSTRACT

Packet filtering firewall is one of the most important mechanisms used by corporations to enforce their security policy. Recent years have seen a lot of research in the area of *firewall management*. Typically, firewalls use a large number of low-level filtering rules which are configured using vendor-specific tools. System administrators start off by writing rules which implement the security policy of the organization. They add/delete/change order of rules as the requirements change. For example, when a new machine is added to the network, new rules might be added to the firewall to enable certain services to/from that machine. Making such changes to the low-level rules is complicated by the fact that the effect of a rule is dependent on its priority (usually determined by the position of the rule in the rule set). As the size and complexity of a rule set increases, it becomes difficult to understand the impact of a rule on the rule set. This makes management of rule sets more error prone. This is a very serious problem as errors in firewall configuration mean that the desired security policy is not enforced.

Previous research in this area has focused on either building tools that generate low-level firewall rules from a given security policy or finding anomalies in the rules, i.e., verifying that the rules implement the given security policy correctly. We propose a technique that aims to infer the high-level security policy from low-level representation. The first step in our approach is that of generating *flattened rules*, i.e., rules without priorities, which are equivalent to the given firewall rule set. Removal of priorities from a rule set enables us to merge a number of rules that have a similar effect. Our rule merging algorithm reduces the size and complexity of the rule set significantly by grouping the *services*, *hosts*, and *protocols* present in these rules into various (possibly overlapping) classes. We have built a prototype implementation[1] of our approach for *iptables* firewall rules. Our preliminary experiments indicate that the technique infers security policy that is at a sufficiently high level of abstraction to make it understandable and debuggable.

Introduction

Firewalls are the first line of defense for protecting corporate networks. System administrators use packet filtering firewalls as one of the mechanisms to implement the security policy of an enterprise. These firewalls are configured using rules that specify matching criteria, and the action to be performed when a packet matches each rule. These rules are matched sequentially against all packets passing through the firewall. These rules can be conflicting, i.e., multiple rules with different actions can match a packet. In such a case, the priority of the rules in the rule set determines the action to be performed. Typically, firewalls use a *first match* policy, i.e., the action corresponding to the first matched rule is taken irrespective of the other rules that can match the packet. Thus the order of rules in a firewall rule set defines a priority relation over the rules. Understanding the effect of firewall rules on network traffic is complicated by this priority relation between the rules.

System administrators initially configure the firewalls with rules that implement the security policy of the organization. As the requirements of the enterprise change, new rules are added or deleted from the rule set without *refactoring*. Over time, the rule set contains many rules which are very similar and the mapping between the security policy and the rules becomes unclear. Managing such large rule sets becomes increasingly difficult leading to configuration errors which are a serious security concern [10]. Hence, firewall management tools become necessary to help system administrators.

Many tools for firewall management (e.g., Firmato [2], Firestarter [3], Shorewall [4]) focus on generating low-level rules from high-level policy language (or GUI). Recent years have seen many works [6, 13, 1] which try to discover configuration errors in the firewalls. But tools which aid in understanding existing firewall rule sets are missing from the arsenal of system administrators. Some tools (e.g., ITVal [8, 9], Fang [7]) provide a way of *querying* whether certain packets will be allowed through the firewall.

The problem with such tools is that the administrator has to know what to query for. Tools like Lumeta Firewall Analyzer [12] try to avoid this problem by automating the task of querying the firewalls. Lumeta Firewall Analyzer queries the firewall for all

[1]This research is supported by NSF grants 0208877 and 0627687.

possible packets that are allowed to pass. For medium to large rule sets this results in a large amount of data being generated. Analyzing such large amounts of data presents another challenge to the system administrator.

We present a novel way to address this problem in this paper. Our approach of inferring the high-level policy from low-level packet filtering rules presents the information to the user in a compact format. Figure 1 shows 24 *iptables* rules taken from a larger firewall rule set (65 rules) being used for a network within our department.[2] It is quite difficult to understand what kind of traffic is allowed through the firewall looking at the script. A new system administrator who is assigned to manage a firewall rule set like this needs to understand the security policy so that she may answer questions such as:

- which services are allowed on each host?
- which hosts are allowed to communicate with each other?
- what protocol is valid between communicating hosts?

Figure 2 shows the 10 rule policy generated by our technique for the same rule set.[3] Clearly it is easier for a system administrator to understand this policy than the rule set in the Figure 1. Moreover, the high-level policy can reveal opportunities for refactoring the low-level rules.

System administrators have an intuitive notion of whether a policy is "complicated" or "simple." The complexity of a policy depends not just on the number of rules in the policy but also on how complicated those rules are. In this work, we define a metric for the complexity of a rule set/policy that captures this notion. This allows us to compare different representations of the same rule sets. Our technique infers policies with low complexity and hence these policies are easier to understand.

Our objective was to develop a technique to infer firewall policies that would help the system administrators to work at a higher level of abstraction. Our technique can be combined with existing techniques to form a comprehensive firewall management toolkit. The benefits of such a toolkit are clear from the following scenario: a system administrator who needs to modify some existing legacy firewall rule set can extract the security policy from the rule set using our technique. She can then make changes to the high-level policy and use an automated tool to generate the low-level rules.

Since our technique uses decision tree like graphs (explained in Section Priority Elimination Phase) to represent the firewall rules, it is easy to enhance our system to provide *querying* facility. Moreover, our system automatically removes redundant rules from the policy. Hence, it is a trivial task to identify such redundant rules in the input rule set using our technique.

We initially present an overview of our approach. The next two sections provide the details of the components in our system. We then discuss related work followed by concluding remarks in final section.

Approach Overview

Our approach for inferring policy consists of two phases. First, in *priority elimination* phase, we convert the low-level rule set that contains rules with priorities to an equivalent rule set that contains rules with no priority relation defined over them. We call the generated rules as *flattened rules*. The flattened rule set

[2]We have modified the IP addresses due to privacy concerns.

[3]Rules in flattened rule set and high-level policy are labeled with alphabets to emphasize the fact that these rules do not have any priority relation defined over them

```
 1.  IPTABLES -A FORWARD -p tcp -d 192.168.1.250 --dport domain -j ACCEPT
 2.  IPTABLES -A FORWARD -p tcp -d 192.168.1.251 --dport smtp -j ACCEPT
 3.  IPTABLES -A FORWARD -p tcp -d 192.168.1.251 --dport smtps -j ACCEPT
 4.  IPTABLES -A FORWARD -p tcp -d 192.168.1.251 --dport imaps -j ACCEPT
 5.  IPTABLES -A FORWARD -p tcp -d 192.168.1.251 --dport pop3s -j ACCEPT
 6.  IPTABLES -A FORWARD -p tcp -d 192.168.1.252 --dport www -j ACCEPT
 7.  IPTABLES -A FORWARD -p tcp -d 192.168.1.126/25 --dport auth -j ACCEPT
 8.  IPTABLES -A FORWARD -s 192.168.1.126/25 -p tcp -d 192.168.1.13 --dport ssh -j ACCEPT
 9.  IPTABLES -A FORWARD -s 192.168.1.126/25 -p tcp -d 192.168.1.14 --dport ssh -j ACCEPT
10.  IPTABLES -A FORWARD -s 192.168.1.126/25 -p tcp -d 192.168.1.15 --dport ssh -j ACCEPT
11.  IPTABLES -A FORWARD -s 192.168.1.126/25 -p tcp -d 192.168.1.20 --dport ssh -j ACCEPT
12.  IPTABLES -A FORWARD -p tcp -d 192.168.1.252 --dport https -j ACCEPT
13.  IPTABLES -A FORWARD -s 192.168.1.254/28 -d 192.168.1.11 -p tcp --dport sunrpc -j ACCEPT
14.  IPTABLES -A FORWARD -s 192.168.1.236 -p tcp -d 192.168.1.35 --dport ipp -j ACCEPT
15.  IPTABLES -A FORWARD -s 192.168.1.254/28 -d 192.168.1.11 -p udp --dport nfs -j ACCEPT
16.  IPTABLES -A FORWARD -s 192.168.1.254/28 -d 192.168.1.11 -p udp --dport 4000:4002 -j ACCEPT
17.  IPTABLES -A FORWARD -p udp -d 192.168.1.251 --dport smtp -j ACCEPT
18.  IPTABLES -A FORWARD -p udp -d 192.168.1.250 --dport domain -j ACCEPT
19.  IPTABLES -A FORWARD -s 192.168.1.254/28 -d 192.168.1.11 -p udp --dport sunrpc -j ACCEPT
20.  IPTABLES -A FORWARD -s 192.168.1.236 -p udp -d 192.168.1.35 --dport ipp -j ACCEPT
21.  IPTABLES -A FORWARD -d 192.168.1.126/25 -p icmp --icmp-type destination-unreachable -j ACCEPT
22.  IPTABLES -A FORWARD -d 192.168.1.126/25 -p icmp --icmp-type parameter-problem -j ACCEPT
23.  IPTABLES -A FORWARD -d 192.168.1.126/25 -p icmp --icmp-type source-quench -j ACCEPT
24.  IPTABLES -A FORWARD -j REJECT
```

Figure 1: Sample *iptables* script.

does not contain any overlapping rules, i.e., there is one and only one flattened rule that can match a given packet. This simplifies the process of inferring policies from rule sets. Unlike the original rules, flattened rules can be arbitrarily reordered without modifying their overall effect. This enables us to reorder and merge similar rules together, thereby reducing the size and complexity of the generated policy.

The problem with priority elimination is that it generates a large number of rules. In the *policy inference* phase, we reduce the number of rules by grouping hosts, services, and protocols into (possibly overlapping) classes and merging rules containing same class of objects. It is not sufficient to produce rule sets with small number of rules as the complexity of the generated rules also affects the complexity of the entire rule set. Arbitrary merging of rules can lead to rule sets which are very complicated. So this phase tries to merge the rules such that the complexity of the inferred policy is minimized. Finally, the inferred policy is presented to the user.

Background

For concreteness, we describe the details of *iptables*. Almost every packet filtering firewall relies on the type of rules used in *iptables*. *Iptables* [14] is the user space command line program used to configure the rule set in the *netfilter* framework in Linux 2.4.x and 2.6.x. *Netfilter* framework enables packet filtering, network address (and port) translation (NA[P]T) and other packet mangling. *Iptables* can be used to configure three independent tables – *filter*, *nat*, and *mangle*, within the kernel. The *filter* table is used to set up rules that are used for filtering packets, while the *nat* table is consulted when a packet that creates a new connection is encountered and the *mangle* for specialized packet alteration.

In this paper, we are concerned only with the *filter* table which is used as a packet filtering firewall. The *filter* table consists of ordered lists of rules that are called *chains*. The order of rules in a chain determines their priority. There are three built-in chains in the *filter* table. INPUT chain is used to filter packets that are destined for the host on which the firewall is running. OUTPUT chain is used to filter packets generated by the firewall host. FORWARD chain is used to filter packets forwarded by the firewall host to other hosts in the network. One powerful feature of *iptables* is that it allows the user to define new chains in addition to the built-in ones. This allows the administrators to group rules which together provide certain high-level function like protecting a subnet.

An *iptables* rule consists of matching criteria and the target. Target specifies the action to be taken when a packet satisfies the matching criterion. Matching criteria is specified in terms of tests on the packet header fields like destination IP address (dhost), source IP address (shost), destination port (dport), source port (sport), protocol (proto). Target can be any of the following: ACCEPT, QUEUE, REJECT, DROP, LOG or name of a user defined chain.

Target ACCEPT means that the packet is to be allowed to pass through the firewall. QUEUE passes the packets on to the user space. For our purposes, semantically QUEUE is similar to ACCEPT as the packet is allowed to reach its destination. So we treat QUEUE just like ACCEPT and omit it from our discussion. DROP and REJECT mean the packet is to be denied. REJECT returns an icmp error packet to the sender while DROP denies the packet without giving any error indication. Target LOG on the other hand just makes an entry to the log file when a matching packet arrives. By specifying user defined chains as targets, conditional call/return semantics can be added to the firewall rule set.

Figure 3 shows a sample *iptables* rule set. All the rules considered are for the FORWARD chain with a default policy of REJECT. Rule 1 specifies that all hosts from the network 192.168.1.0/24 are allowed to connect to the host 120.240.18.1 using SMTP. Rule 2 specifies that host 120.240.18.1 can connect to the network 120.240.20.0/24 using SMTP. This can be a real world scenario where 192.168.1.0/24 is an internal network of an organization, 120.240.20.0/24 is external network and 120.240.18.1 is the SMTP server for that organization. The rule set says that SMTP server can send SMTP traffic to external network and internal hosts can send SMTP traffic to SMTP server but not to the external network.

We represent the *iptables* rules in tabular format for ease of understanding. Table 1 is the tabular representation of the rules in Figure 3. We list all rules in a table in the order of their priority. The columns indicate the packet fields being tested. A rule is represented as a row with values for the packet fields being

Allow only the following packets:

```
  a. tcp, udp FROM 192.168.1.254/28 TO 192.168.1.11 FOR sunrpc
  b. udp FROM 192.168.1.254/28 TO 192.168.1.11 FOR nfs, ports [4000-4002]
  c. tcp FROM 192.168.1.126/25 TO [192.168.1.13 - 15], 192.168.1.20 FOR ssh
  d. tcp, udp FROM 192.168.1.236 TO 192.168.1.35 FOR ipp
  e. tcp TO 192.168.1.126/25 FOR auth
  f. icmp TO 192.168.1.126/25 OF TYPES destination-unreachable, parameter-problem, source-quench
  g. tcp, udp TO 192.168.1.250 FOR domain
  h. tcp, udp TO 192.168.1.251 FOR smtp
  i. tcp TO 192.168.1.251 FOR smtps, imaps, pop3s
  j. tcp TO 192.168.1.252 FOR www, https
```

Figure 2: Higher level policy for rules in Figure 1.

tested filled in the respective columns. If a rule does not contain any test on a particular field, then that column has a wild-card character "*". A "*" for a field indicates that any value of the field will match this rule. The action associated with a rule is shown in the last column. Note that we omit many fields like icmp-type from examples to avoid clutter.

Even though our technique can be applied to chains with default ACCEPT policy, all examples and discussions assume that the chains have default REJECT policy. Note that the chain name is shown above the rules in the tabular format to make the tables more understandable. In practice, we generate different rule sets for different built-in chains.

Priority Elimination Phase

In this phase we take the rule set with priorities and generate *flattened* rule set. Flattened rule set contains rules which have no priority relation so they can be arbitrarily reordered and merged to generate a compact policy. The idea behind flattening of rules is simple. Consider a rule set *RS* with rules R_i such that priority of R_i is higher than the priority of R_j iff $i < j$. The semantics of such a *prioritized* rule set are that a packet is matched by a rule R_j iff it satisfies the matching criteria of R_j and doesn't satisfy the matching criteria of any of the rules R_i such that $i < j$. In other words, a packet can match a rule only if it is not matched by any higher priority rule. For example, R_3 can match a packet only if R_1 and R_2 do not match it. Thus a prioritized rule set can be converted to a flattened rule set by replacing R_j by $\bigwedge_{i=1}^{j-1} \neg R_i \wedge R_j, 2 \le j \le n$ where n is the total number of rules in the set. In our example, R_1 will not be modified while R_2 will be replaced by $\neg R_1 \wedge R_2$ and R_3 by $\neg R_1 \wedge \neg R_2 \wedge R_3$.

The problem with the naive way of generating flattened rules is that it can lead to exponential number of rules. In [11], we developed a way of creating a directed acyclic graph (DAG) called *packet classification automaton* that avoids this exponential blowup. We use the packet classification automaton to generate flattened rule set. Here we describe the characteristics of the automaton without going into the details of the construction algorithm; which can be found in [11].

- each node (except the final nodes) is annotated with a packet header field. This denotes that the node performs a test on that field.
- the leaf nodes correspond to the action to be performed when a packet is matched by a rule. For example, for *iptables* the leaf nodes correspond to the higher level actions allow and deny. We map targets like ACCEPT, QUEUE, and LOG to allow and REJECT and DROP to deny. A path from the root to a leaf represents the tests to be performed on a packet to match it against the rule set and the action to be taken on the packet.
- at each node the outgoing edges are labeled with the different values specified for the packet header field specified on the node.
- at each node there is an additional outgoing edge called as "else" edge. This edge is taken when a packet has a field value different from any of the values listed in the other outgoing edges from that node.
- nodes at the same height in different subgraphs can have tests for different fields.

This automaton has the following interesting properties:

- **Property 1** *Packet classification automaton is equivalent to the prioritized rule set, i.e., any packet that is allowed/denied by the rule set has a path from root to allow/deny node in the automaton and vice versa.*
- **Property 2** *If the input rule set is comprehensive, i.e., for every packet there is a rule in the rule set that matches it, then the automaton has a path from root to a leaf for every packet. Moreover, the path from root to leaf is unique.*

We generate flattened rule set by considering all paths from root to the leaves in the graph. Each path corresponds to a rule in the flattened rule set. Consider a *iptables* script with the three rules for FORWARD chain shown in Listing 1.

```
1. IPTABLES -A FORWARD -s 192.168.1.0/24 -d 120.240.18.1 --dport 25 -j ACCEPT
2. IPTABLES -A FORWARD -s 120.240.18.1 -d 120.240.20.0/24 --dport 25 -j ACCEPT
3. IPTABLES -A FORWARD -j REJECT
```

Figure 3: *iptables* rule set 1.

#	shost	sport	dhost	dport	target
FORWARD (Default: Reject)					
1	192.168.1.0/24	*	120.240.18.1	25	**ACCEPT**
2	120.240.18.1	*	120.240.20.0/24	25	**ACCEPT**

Table 1: *iptables* rules in Figure 3 represented in tabular format.

```
-d 192.168.1.1 -s 192.168.1.3 --dport 22 -j ACCEPT
-d 192.168.1.2 -s 192.168.1.4 --dport 22 -j ACCEPT
-j REJECT
```

Listing 1: Sample *iptables* rule set as input.

Figure 6 shows the packet classification automaton for this sample rule set. Here each node is labeled with the packet field being tested at that node.

Pruning

Figure 4 shows the packet classification automaton for a firewall rule set with 65 rules for a small network within our department. The bottom row has two leaf nodes corresponding to the actions: allow and deny. We can see that even for small sized rule set, there are a large number of paths in the graph.

We prune this graph to reduce the number of paths that we have to consider. The following are the steps that we perform for pruning this graph.

1. We remove deny node and all incoming edges to it. If this results in an intermediate node becoming leaf node, then we remove that node and its incoming edges. We recursively do this for all *ancestors* of deny except the root node. Figure 7 shows the graph after deny node has been removed from the graph in Figure 6. Now the graph contains only paths from root to allow. This means that the flattened rule set that we generate from this graph is no longer *comprehensive*. But this problem can be easily solved by having a default reject policy for the flattened rules, i.e., packets that are not matched by any rule in the flattened rule set are discarded.
2. We do a bottom-up traversal of the graph and merge equivalent states. We consider two states r and s as equivalent if they have transitions to the same state t_i on label l_i for all outgoing edges. For the graph in Figure 7, both the dport nodes have an edge to allow for value 22. These nodes are merged to get a graph as shown in Figure 8.
3. The previous two steps create new opportunities for reducing the number of paths though the graph. We can now merge multiple edges which connect the same nodes. In the case where the edge merging involves else edge, the merged edge is labeled with else. A special case of this is when all outgoing edges from a node can be merged with else edge. In this case the node with the merged outgoing edges is removed as the merged edge indicates that this

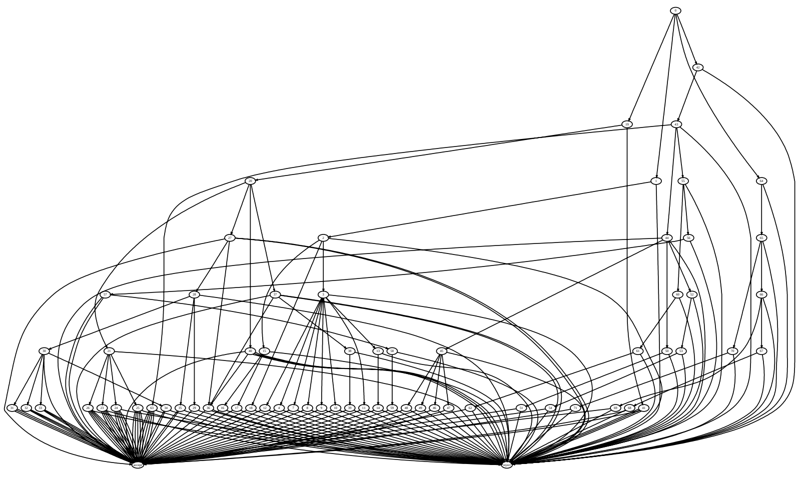
Figure 4: Initial graph for network in the department.

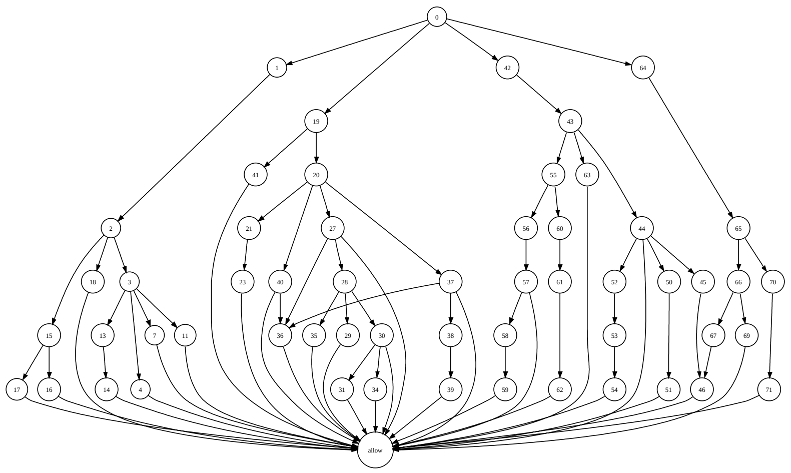

Figure 5: Pruned graph for graph in Figure 4.

transition is taken irrespective of the value of the field in the removed node. This edge merging is also done in a bottom up fashion.

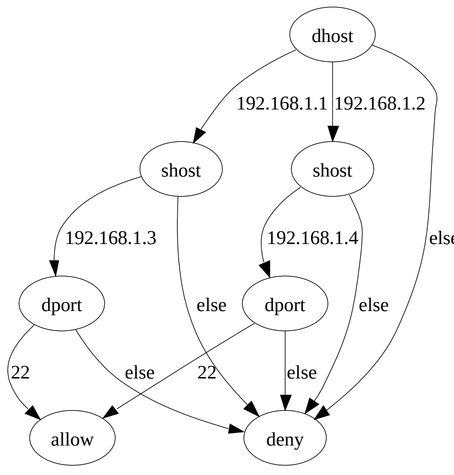

Figure 6: Unpruned graph for sample rules.

Figure 5 shows the pruned graph corresponding to the graph in Figure 4. We can read off all the paths from the root to accept to get flattened rule set.

Policy Inference Phase

As the main goal of our research was to come up with a compact representation of the rules, we asked ourselves the following questions:

- how can we compare two representations of the same rule set?
- how can we reduce the complexity of the flattened rules?

In this section we present our answers to these questions.

Complexity

We call a particular representation of rules as *policy*. Intuitively, a policy that requires a bigger description is more complex. For example, consider an organization which has 192.168.1.0/24 as the internal network. It has rules that allow auth packets to all hosts in the network and ssh packets only to the hosts 192.168.1.5, 192.168.1.6, and 192.168.1.7. These rules can be represented as *Policy 1* as shown in Listing 2. A more compact way of representing these rules (*Policy 2*) is shown in Listing 3.

We capture this notion of how complicated a policy is by the following definition:

Definition 3: Complexity of a policy

- **Data item** *is any value that is used in a rule.*
- **Complexity of a rule** *is the number of data items present in the rule.*
- **Complexity of a policy** *is the sum of the complexity of all rules in the policy.*

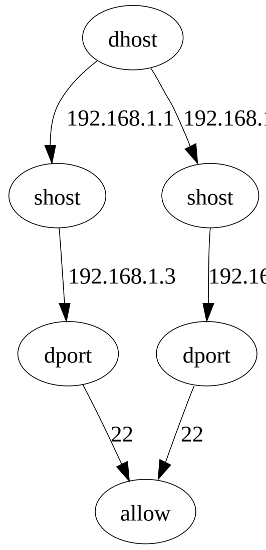

Figure 7: Graph with deny node removed from the graph in Figure 6.

Data items refer to the different values of packet fields present in the policy. For example, in the policy given above, the data items are 192.168.1.0/24, [192.168.1.5 -| 7], auth, and ssh. For the earlier policy the data items are [192.168.1.0 - 4], [192.168.1.8 - 255], [192.168.1.5 - 7], auth, and ssh. The complexity of rules in *Policy 1* is 3 for rule *a* and 3 for rule *b*. We note that ranges such as [192.168.1.5 - 7] are treated as a single data item and hence contribute 1 to the complexity of a rule. Even though our examples contain * for certain fields, they are there just to increase readability. Our final policy (as shown in Figure 2) does not contain any "*". Therefore, "*" doesn't contribute anything to the complexity of a rule. Similarly, in *Policy 2* the complexity of rules *a* and *b* are 2 each. The complexity of *Policy 2* is 4. This is less than that of *Policy 1* which is 6. Hence we can say that *Policy 2* is a more compact representation than *Policy 1*.

Problem Statement

We describe the problem statement in this section. At the end of *priority elimination* phase we have

```
Accept packets
a. TO [192.168.1.0 - 4], [192.168.1.8 - 255] FOR auth
b. TO [192.168.1.5 - 7]  FOR auth, ssh
```

Listing 2: Policy 1.

```
Accept packets
a. TO 192.168.1.0/24 FOR auth
b. TO [192.168.1.5 - 7] FOR ssh
```

Listing 3: Policy 2 – More compact version of policy in Listing 2.

a large number of rules. We want to merge the rules in such a way that the complexity of the generated rule set is minimum. We can merge rules which have the same values for all but one field by performing a *union* operation over the field which has differing values in the rules. The main issue with merging rules is that different subsets of rules can be merged on different fields. So we need to select the subsets in such a way that merging them leads to minimum complexity of the entire rule set. This problem can be illustrated by considering the rule set shown in Table 2. We can merge rules *b* and *c* on shost to get a new rule $\{bc\}$. Note that we label a merged rule by concatenating the labels of the original rules. We enclose the new label in *braces* to indicate how the rules merged.[4] We can then merge $\{bc\}$ with *a* to get rule $\{a\{bc\}\}$. This gives the following policy (*Policy 3*) which has a complexity of 12; see Listing 4.

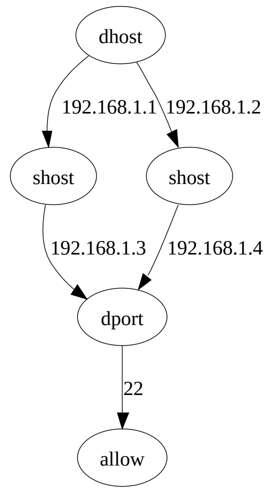

Figure 8: dport node merged from graph in Figure 7.

[4]The notation $\{bc\}$ is different from $\{b,\ c\}$ which means a set containing the rules *b* and *c*

The complexity of each rule is written in parenthesis besides the rules. Since all the rules in the flattened rule set have the same action, we can even generate overlapping rules to get a more compact policy. We can merge rules *a*, *b*, and *c* as before to get rule $\{a\{bc\}\}$, and also merge *c* and *d* on dhost followed by $\{cd\}$ with *e* on dport. Listing 5 shows policy (*Policy 4*), with complexity 10, that is obtained by merging the rules in this way.

The Policy Complexity Definition is tied to the way the rules are represented. Thus, the interesting question is *given a rule set, can we find an equivalent representation which enforces the same security policy but has lower complexity?* Our goal is to find a *representation of the rule set that implements the same policy and has minimum complexity*. A natural way to select the policy with minimum complexity is to assign weights based on the complexity to all subsets of the flattened rule set and select a minimum weight set cover. The problem of determining the minimum weight set cover is NP-complete. However, we have found that fairly simple methods, based on exploiting the structure of the flattened rules, yield good results for finding a policy with low complexity.

Computing Weight of Subsets of Rules

Computing the complexity of all subsets of a rule set is also very hard. To see this, consider that we have a set of *n* rules. We want to find a policy with minimum complexity for this rule set. To find the complexity of a set containing $n-1$ of these rules is similar to the original problem.

In practice, we can avoid this problem as we do not need to generate all subsets of rules in the original rule set. To understand this consider original rule set $\{a,\ b,\ c\}$ such that rules *a* and *b* can be merged on certain field. Now the *weight* of $\{a,b\}$ is the complexity of the merged rule $\{ab\}$. But the *weight* of $\{b,c\}$ is the sum of the complexities of *b* and *c*. So we do not need

```
Accept packets
{a{bc}}. 192.168.1.1, 192.168.1.10 TO 192.168.2.1 FOR http, smtp          (5)
d.       192.168.1.10 TO 192.168.2.2 FOR smtp                             (3)
e.       192.168.1.10 TO 192.168.2.1, 192.168.2.2 FOR ssh                 (4)
```

Listing 4: Policy 3.

#	shost	dhost	dport	target
a	192.168.1.1, 192.168.1.10	192.168.2.1	80	ACCEPT
b	192.168.1.1	192.168.2.1	25	ACCEPT
c	192.168.1.10	192.168.2.1	25	ACCEPT
d	192.168.1.10	192.168.2.2	25	ACCEPT
e	192.168.1.10	192.168.2.1, 192.168.2.2	22	ACCEPT

Table 2: Sample flattened rule set.

```
Accept packets
{a{bc}}. 192.168.1.1, 192.168.1.10 TO 192.168.2.1 FOR http, smtp         (5)
{{cd}e}. 192.168.1.10 TO 192.168.2.1, 192.168.2.2 FOR smtp, ssh          (5)
```

Listing 5: Policy 4.

to consider the *weight* of $\{b,c\}$ if we have the *weights* for $\{b\}$ and $\{c\}$. This leads us to conclude that we need to consider only the subsets of the original set in which each rule merges with some other rule or with a new merged rule. We maintain the subsets of rules that we need to consider in a *working set*.

Here we describe the algorithm shown in Figure 9 to generate the working set. Initially we have a set $\mathcal{R}$ of flattened rules $\{r_1, r_2, \ldots, r_n\}$. We want to generate a set $\mathcal{W}$ that contains the subsets of $\mathcal{R}$ that we need to consider for the minimum weight set cover problem. We start by putting all singleton subsets of $\mathcal{R}$ in $\mathcal{W}$. For example in Figure 2,

$$\mathcal{W} = \{\{a\}:4, \{b\}:3, \{c\}:3, \{d\}:3, \{e\}:4\}.$$

Note that the value after ":" is the *weight* of the corresponding set. For singleton sets, the *weight* is the same as the complexity of the rule. Now we compare each element in $\mathcal{W}$ with the other elements in $\mathcal{W}$. If the two elements under consideration can be merged, then we merge them and add the new merged rule to the working set. Now,

$$\mathcal{W} = \{\{a\}:4, \{b\}:3, \{c\}:3, \{d\}:3, \{e\}:4, \{bc\}:4, \{cd\}:4\}.$$

We continue this process of merging and adding new rules till no more rules can be added to $\mathcal{W}$. In our example, $\{bc\}$ can be merged with $\{a\}$ and $\{cd\}$ with $\{e\}$. After this no more merging is possible. So the final set is

$$\mathcal{W} = \{\{a\}:4, \{b\}:3, \{c\}:3, \{d\}:3, \{e\}:4, \{bc\}:4, \{cd\}:4, \{a\{bc\}\}:5, \{\{cd\}e\}:5\}.$$

Merge Graphs

Comparing each element in $\mathcal{W}$ with all the other elements is computationally expensive. We overcome this problem by generating a graph, which we call as *merge graph*, that allows us to avoid many comparisons. *Merge graph* is an acyclic graph which initially contains nodes corresponding to the flattened rules and no edges. As we merge rules, we add nodes corresponding to the merged rules and edges from the new node to the constituent rule nodes. For example, after we add $\{bc\}$ to $\mathcal{W}$, we can represent $\mathcal{W}$ as a *merge graph* as shown in Figure 10. The *merge graph* may contain disconnected subgraphs. For each node in the graph, we compare it with only the nodes in other disconnected components. This way we avoid redundant comparisons. In Figure 10, we compare the rule $\{bc\}$ with $\{a\}$, $\{d\}$, and $\{e\}$. This avoids redundant comparisons with $\{b\}$ and $\{c\}$. For large rule sets with multiple rules that merge, this optimization proves very useful. Figure 11 shows the final *merge graph* for our example.

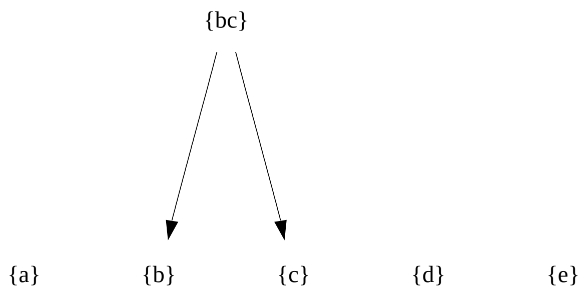

Figure 10: Merge graph after adding {bc}.

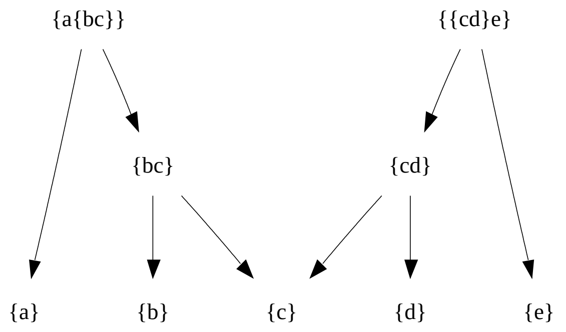

Figure 11: Final merge graph.

Solving Minimum Weight Set Cover

Now that we have the working set we can find the minimum weight set cover to get a policy with low complexity. Conceptually we can think of each element of $\mathcal{W}$ as a set containing the rules that have been merged to form that element. For the example in the previous section,

```
procedure GenerateWorkingSet($\mathcal{R}$) {
    $\mathcal{W} = \phi$
    for $i$ = 1 to $|\mathcal{R}|$ do              /* $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ */
        $\mathcal{W} = \mathcal{W} \cup \{r_i\}$              /* add all flattened rules to the working set */
    end
    for $i$ = 1 to $|\mathcal{W}|$ do
        for $j$ = 1 to $|\mathcal{W}|$ do
            $S$ = MergeRules($w_i$, $w_j$)   /* $w_i$, $w_j \in \mathcal{W}$ */
            if $S \neq \phi$
                $\mathcal{W} = \mathcal{W} \cup S$       /* add the new merged rule to the working set */
            endif
        end
    end
    return $\mathcal{W}$
}
```

Figure 9: Algorithm for Constructing Working Set.

$$\mathcal{W} = \{\{a\}{:}4, \{b\}{:}3, \{c\}{:}3, \{d\}{:}3, \{e\}{:}4, \{b,c\}{:}4, \{c,d\}{:}4, \{a,b,c\}{:}5, \{c,d,e\}{:}5\}\ .$$

Our target is to find a set cover $C = \{c_1, c_2, \ldots, c_k\}$, i.e., $C \subseteq \mathcal{W} \wedge \bigcup_{i=1}^{k} c_i = \mathcal{R}$ such that $\sum_{i=1}^{k} weight(c_i)$ is minimum.

Figure 12 shows our algorithm based on greedy heuristics to solve the problem. We use a set, $\mathcal{A}$, to keep track of the rules that are covered as the set cover C is built. For each set in $\mathcal{W}$, we define,

$$cost(w_i) = \frac{weight(w_i)}{|w_i / A|}$$

where $cost(w_i)$ represents the cost incurred (in terms of *weight*) per new rule that will be covered by including a set w_i in the set cover. In each iteration, we pick a set w_i that has the lowest cost (step 6 in our algorithm). In our example, initially the *cost* is 4/1 = 4 for $\{a\}$, 4/2 = 2 for $\{b, c\}$, 5/3 = 1.67 for $\{a, b, c\}$ and so on. In case of a tie, we pick the set with higher cardinality. This algorithm returns $C = \{\{a, b, c\}, \{c, d, e\}\}$ as the minimum weight set cover. We know that these sets correspond to merged rules $\{a\{bc\}\}$ and $\{\{cd\}e\}$. So we can represent the rules in the example as *Policy 4* as shown in the previous section.

Related Work

There are many tools (e.g., Firmato [2], Shorewall [4], Firestarter [3]) that are available for generating low-level firewall rules from high-level policy. Our technique can be used in conjunction with these tools to help refactoring. These tools can be used to generate firewall rules from scratch. If our technique is combined with these tools then we can use these tools to make changes to existing low level rules.

Fang [7] and ITVal [9] are tools that provide querying facility. But it puts the onus on the system administrator to figure out what queries to perform. Lumeta Firewall Analyzer [12] solves this problem by querying the system for all packets that can be accepted. The problem with this approach is that this results in a large amount of data being presented to the user. In contrast, we try to present the result of our analysis in a compact fashion to the administrator. Moreover, it is easy to provide querying capability using our technique.

Yuan, et al. [13], Gouda, et al. [6], Al-Shaer, et al. [1] have looked at the problem of identifying configuration errors in the firewall rules. The problem with these approaches is that the administrator has to decide whether the alert generated by these tools are due to rules that are put in intentionally or unintentionally. Our technique can help in solving this problem by providing a high level view of the security policy.

Marmorstein, et al. [8] generate policy by grouping similar hosts. Our work is more general in the sense that we can group together arbitrary things to generate more compact representation. Golnabi et al. [5] have looked at the problem of generating high level policy. But their approach is based on data mining of firewall logs while we try to extract the policy from the rules itself.

Conclusions

In this paper, we presented a new technique for extracting high-level security policy from low-level rules. Unlike previous techniques, our technique generates policy which is compact. This will help system administrators to understand the existing low level rule sets and encourage them to refactor the low-level rules instead of making small changes to the rules set when requirements change. This will make the rule sets more manageable and will likely result in reducing the errors in configuration of firewalls. We also presented a way for comparing whether one policy representation is better than another. In our preliminary experiments, we obtained 50 flattened rules with 197 data items after the *priority elimination* phase from a 65 rule firewall. The final inferred policy on the other hand had just 21 rules with 129 data items. These results indicate that we can generate high level policy which is easier to understand and manage.

Acknowledgments

We would like to thank Mayur Mahajan for his help in developing the graph library used for generating

```
 1.  procedure MinimumWeightSetCover($\mathcal{R}$, $\mathcal{W}$) {
 2.      $\mathcal{A} = \phi$                 /* $\mathcal{A} \subseteq \mathcal{R}$ is the set of covered elements */
 3.      $C = \phi$                 /* $C \subseteq \mathcal{W}$ is the solution set */
 4.      while $\mathcal{A} \neq \mathcal{R}$ do          /* $\mathcal{A}$ is not a set cover */
 5.          for $i = 1$ to $|\mathcal{W}|$
 6.              compute $cost(w_i)$
 7.          end
 8.          choose $w_i$ with minimal $cost(w_i)$
 9.          $\mathcal{A} = \mathcal{A} \cup w_i$          /* rules in $w_i$ are covered */
10.          $C = C \cup \{w_i\}$         /* add $w_i$ to solution set */
11.          $\mathcal{W} = \mathcal{W} / \{w_i\}$        /* remove $w_i$ from working set */
12.      end
13.      return $C$
14.  }
```

Figure 12: Algorithm for Minimum Weight Set Cover.

the flattened rules and for his work on pruning the graphs. We would like to thank Weiqing Sun, Lorenzo Cavallaro, and the anonymous reviewers for their insightful comments.

This material is based upon work supported by the NSF grants 0627687 and 0208877. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

Author Biographies

All The authors of this paper are members of the Secure Systems Laboratory of Stony Brook and their homepages are accessible on the web from the laboratory page at http://www.seclab.cs.sunysb.edu .

R. Sekar is currently Professor of Computer Science and heads the Secure Systems Laboratory at Stony Brook University. Prof. Sekar's research interests include computer system and network security, software and distributed systems, programming languages and software engineering. He can be reached by email at sekar@cs.sunysb.edu .

Alok Tongaonkar is a Ph.D. student in the CS department at Stony Brook. His main research area is computer security and is currently working on analyzing and optimizing firewalls and intrusion detection systems. He is available at alok@cs.sunysb.edu .

Niranjan Inamdar is a M.S. student in the CS department at Stony Brook. Niranjan does research in the area of computer security. He can be reached via email at niranjan@cs.sunysb.edu .

Bibliography

[1] Al-Shaer, Ehab and Hazem Hamed, "Discovery of Policy Anomalies in Distributed Firewalls," *IEEE INFOCOM*, 2004.

[2] Bartal, Yair, et al., "Firmato: A Novel Firewall Management Toolkit," *IEEE Security and Policy*, 1999.

[3] *Firestarter*, http://www.fs-security.com .

[4] *Shorewall Firewall*, http://www.shorewall.net .

[5] Golnabi, Korosh, et al., "Analysis of Firewall Policy Rule using Data Mining Techniques," *IEEE/IFIP Network Operations and Management Symposium*, 2006.

[6] Gouda, Mohamed G. and Alex X. Liu, "Firewall Design:Consistency, Completeness, and Compactness," *International Conference on Distributed Computing Systems*, 2004.

[7] Mayer, Alain, et al., "Fang: A Firewall Analysis Engine," *IEEE Symposium on Security and Privacy*, 2000.

[8] Marmorstein, Robert and Phil Kearns, "Firewall Analysis with Policy-Based Host Classification", *20th Large Installation Systems Administration Conference*, 2006.

[9] Marmorstein, Robert and Phil Kearns, "A Tool for Automated Iptables Firewall Analysis," *Freenix Track, USENIX Annual Technical Conference*, 2005.

[10] Rubin, Aviel, Dan Geer, and Marcus Ranum, *Web Security Sourcebook*, Wiley Computer Publishing, 1997.

[11] Tongaonkar, Alok, "Fast Pattern-Matching Techniques for Packet Filtering," *Master's Thesis Report, Stony Brook University*, 2004, http://www.seclab.cs.sunysb.edu/seclab/pubs/theses/alok.pdf .

[12] Wool, Avishai, "Architecting the Lumeta Firewall Analyzer," *10th USENIX Security Symposium*, 2001.

[13] Yuan, Lihua, et al., "Fireman: A Toolkit for Firewall Modeling and Analysis," *IEEE Symposium on Security and Privacy*, 2006.

[14] Eychenne, Herve, *iptables Man Page*, March, 2002.