

Firewall Policy Advisor for Anomaly Detection, Rules Editing and Translation

Ehab S. Al-Shaer and Hazem H. Hamed

*Multimedia Networking Research Laboratory
School of Computer Science, Telecommunications and Information Systems
DePaul University, 243 S Wabash Ave, Chicago, IL 60604
Tel: (312)362-5137
{ehab, hhamed}@cs.depaul.edu*

Abstract

Firewalls are core elements in network security. However, managing firewall rules, especially for enterprise networks, has become complex and error-prone. Firewall filtering rules have to be carefully written and organized in order to correctly implement the security policy. In addition, inserting or modifying a filtering rule requires thorough analysis of the relationship between this rule and other rules in order to determine the proper order of this rule and commit the updates. In this paper, we present a set of techniques and algorithms that provide (1) automatic anomaly detection for discovering rule conflicts and potential problems in legacy firewalls, (2) anomaly-free policy editing for rule insertion, modification and removal, and (3) concise translation of filtering rules to high-level textual description for user visualization and verification. This is implemented in a user-friendly tool called “Firewall Policy Advisor.” The firewall policy advisor significantly simplifies the management of any generic firewall policy written as filtering rules, while minimizing network vulnerability due to firewall rule misconfiguration.

Keywords: firewall, security management, security policy, policy conflict.

Firewall Policy Advisor for Anomaly Detection, Rules Editing and Translation

Ehab S. Al-Shaer and Hazem H. Hamed
Multimedia Networking Research Laboratory
School of Computer Science, Telecommunications and Information Systems
DePaul University, Chicago, IL 60604
{ehab, hhamed}@cs.depaul.edu

Abstract

Firewalls are core elements in network security. However, managing firewall rules, especially for enterprise networks, has become complex and error-prone. Firewall filtering rules have to be carefully written and organized in order to correctly implement the security policy. In addition, inserting or modifying a filtering rule requires thorough analysis of the relationship between this rule and other rules in order to determine the proper order of this rule and commit the updates. In this paper, we present a set of techniques and algorithms that provide (1) automatic anomaly detection for discovering rule conflicts and potential problems in legacy firewalls, (2) anomaly-free policy editing for rule insertion, modification and removal, and (3) concise translation of filtering rules to high-level textual description for user visualization and verification. This is implemented in a user-friendly tool called “Firewall Policy Advisor.” The firewall policy advisor significantly simplifies the management of any generic firewall policy written as filtering rules, while minimizing network vulnerability due to firewall rule misconfiguration.

Keywords: firewall, security management, security policy, policy conflict.

1. Introduction

With the global Internet connection, network security has gained significant attention in the research and industrial communities. Due to the increasing threat of network attacks, firewalls have become important integrated elements not only in enterprise networks but also in small-size and home networks. Firewalls have been the frontier defense for secure networks against attacks and unauthorized traffic by filtering out unwanted network traffic coming into or going from the secured network. The filtering decision is taken according to a set of ordered filtering rules defined based on predefined security policy requirements.

Although deployment of firewall technology is an important step toward securing our networks, the complexity of managing firewall rule policy might limit the effectiveness of firewall security. When the filtering rules are defined, serious attention has to be given to rule relations and interactions in order to determine the proper rule ordering and guarantee correct security policy semantics. As the number of filtering rules increases, the difficulty of writing a new rule or modifying an existing one also increases. It is very likely; in this case, to introduce conflicting rules such as rules having the same filtering part but different actions, one general rule shadowing another specific related rule, or correlated rules whose relative ordering determines different actions for the same packets. In addition, a typical large-scale enterprise network might involve hundreds of rules that might be written by different administrators in various times. This significantly increases the potential of anomalies (conflicts) in the firewall rules and makes the network more vulnerable.

Therefore, the effectiveness of firewall security is dependent on providing policy management techniques/tools that enable network administrators to analyze, purify and verify the correctness of written firewall legacy rules. In this paper, we define a formal model for firewall rule relations and their filtering representation. The proposed model is simple and visually comprehensible. We use this model to develop an anomaly discovery algorithm to report any anomaly that exists among the filtering rules. We

then develop anomaly-free firewall rule editing, which greatly simplifies adding and modifying rules into firewall policy. We finally develop a policy translator that gives a concise textual description of the entire policy rules for user verification. We used the Java programming language to implement these algorithms in one graphical user-interface tool called “Firewall Policy Advisor.”

Although firewall security has been given strong attention in the research community, the emphasis was mostly on the filtering performance and hardware support issues [5, 8, 10, 11, 17]. On the other hand, few related work [6, 10] present a resolution for the correlation conflict problem only. Other approaches [2, 9, 12, 14, 18] propose using a high-level policy language to define and analyze firewall policies and then mapping this language to filtering rules. Firewall query-based languages based on filtering rules are also proposed in [7, 11]. So in general, we consider our work a new progress in this area because it offers new techniques for complete anomaly detection, rules editing and translation that can be used on legacy firewall policies of low-level filtering rule representation.

This paper is organized as follows. In Section 2, we give an introduction to firewall operation and filtering rule format. In Section 3, we formally define filtering rule relations, and we present our proposed model of filtering rule relations and the policy tree representation. In Section 4, we classify and define firewall policy anomalies, and then we describe the anomaly detection algorithm and implementation. In Section 5, we present the design and implementation of anomaly-free firewall rules editor. In Section 6, we present the firewall policy translator. In Section 7, we give a summary of related work. Finally, in Section 8, we show our conclusions and our future work plan.

2. Firewall Background

A firewall is a network element that controls the traversal of packets across the boundaries of a secured network based on a specific security policy. A firewall security policy is a list of ordered filtering rules that define the actions performed on matching packets. A rule is composed of filtering fields (also called network fields) such as protocol type, source IP address, destination IP address, source port and destination port, and a filter action field. Each network field could be a single value or range of values. Filtering actions are either to *accept*, which passes the packet into or from the secure network, or to *deny*, which causes the packet to be discarded. The packet is accepted or denied by a specific rule if the packet header information matches all the network fields of this rule. Otherwise, the next following rule is used to test the matching with this packet again. Similarly, this process is repeated until a matching rule is found or the default policy action is performed. In this paper, we assume a “deny” default policy action.

Filtering Rule Format. It is possible to use any field in IP, UDP or TCP headers in the rule filtering part, however, practical experience shows that the most commonly used matching fields are: protocol type, source IP address, source port, destination IP address and destination port. Some other fields, like TTL and TCP flags, are occasionally used for specific filtering purposes. The following is the common format of packet filtering rules in a firewall policy:

```
<order> <protocol> <src ip> <src port> <dst ip> <dst port> <action>
```

In this paper, we refer to the fields in the shaded box as “5-tuple filter” or “network fields”, interchangeably. The *order* of the rule determines its position relative to other filtering rules. The *protocol* specifies the transport protocol of the packet, and can be one of these values: IP, ICMP, IGMP, TCP or UDP. The *src_ip* and *dst_ip* specify the IP addresses of the source and destination of the packet respectively. The IP address can be a host (e.g., 140.192.37.120), or a network address (e.g., 140.192.37.*). The *src_port* and *dst_port* fields specify the port address of the source and destination of the packet respectively. The port can be either a single specific port number, or any port number indicated by “any”. As an example, the following security policy is to *block all TCP traffic coming from the network 140.192.37.* except HTTP*:

```
1: tcp, 140.192.37.*, any, *.*.*.*, 80, accept
2: tcp, 140.192.37.*, any, *.*.*.*, any, deny
```

Some firewall implementations allow the usage of non-wildcard ranges in specifying source and destination addresses or ports. However, it is always possible to split a filtering rule with a multi-value port field into several rules each with a single-value port field. In this paper, we use only wildcard ranges. We also assume a statefull firewall; i.e. if a TCP connection is accepted, all incoming and outgoing packets that belong to the connection are accepted as well.

3. Firewall Policy Modeling

As a basic requirement for any firewall policy management solution, we first modeled the relations and the representation of firewall rules in the policy. This model is complete (i.e., means includes all rules possible in any firewall policy) and efficient (i.e., means easy to implement and easy to use). Rule relation modeling is necessary for analyzing firewall policy and designing management techniques such as conflict detection and rules editing. The rules or policy representation modeling is important for implementing these management techniques and visualizing the firewall policy structure. In this section, we describe formally our model of firewall rule relations and policies.

3.1. Formalization of Firewall Rule Relations

To be able to build a useful model for filtering rules, we need to determine all the relations that may relate two or more packet filters. In this section we define all the possible relations that may exist between filtering rules, and we prove that there is no other relation exists. We determine the relations based on comparing the network fields of filtering rules. The values of the same field in two different rules may be equal, inclusive or distinct. Two values are *equal* if they exactly match, and are *inclusive* if one value is a subset of, and not equal, the other (superset), and are *distinct* otherwise. Two fields *match* if they are equal or inclusive. For examples, a source address value of 140.192.37.10 matches 140.192.37.* and does not match 140.192.37.20.

Definition 1: Rules R_x and R_y are *exactly matched* if every field in R_x is equal to the corresponding field in R_y . Formally:

$$R_x \text{ exactly matches } R_y \text{ iff} \\ \forall i: R_x[i] = R_y[i] \text{ where } i \in \{\text{protocol, src_ip, src_port, dst_ip, dst_port}\}$$

For example, rule 1 and rule 2 below are exactly matched since all corresponding fields in both rules are equal.

```
1: tcp, 140.192.37.10, any, 163.122.51.*, 21, accept
2: tcp, 140.192.37.10, any, 163.122.51.*, 21, deny
```

Definition 2: Rules R_x and R_y are *inclusively matched* if they do not exactly match and if every field in R_x is a subset or equal to the corresponding field in R_y . Formally:

$$R_x \text{ inclusively matches } R_y \text{ iff} \\ \forall i: R_x[i] \subseteq R_y[i] \text{ and } \exists j \text{ such that: } R_x[j] \neq R_y[j] \\ \text{where } i, j \in \{\text{protocol, src_ip, src_port, dst_ip, dst_port}\}$$

In this relation, R_x is called the *subset match* while R_y is called the *superset match*. For example, rule 1 and rule 2 below are inclusively matched since they do not exactly match and every field in rule 1 is a subset or equal to the corresponding field in rule 2. Rule 1 is the subset match of the relation while rule 2 is the superset match.

```
1: tcp, 140.192.37.10, any, 163.122.51.*, 80, accept
2: tcp, 140.192.37.*, any, 163.122.51.*, any, deny
```

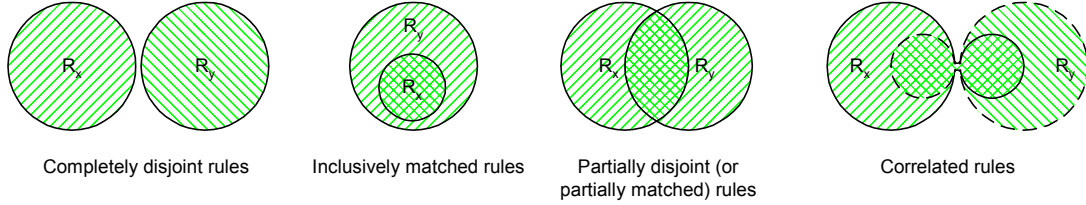


Figure 1. Relations between two filtering rules R_x and R_y .

Definition 3: Rules R_x and R_y are *completely disjoint* if every field in R_x is not a subset and not a superset and not equal to the corresponding field in R_y . Formally:

R_x and R_y are *completely disjoint* iff

$$\forall i: R_x[i] \equiv R_y[i] \quad \text{where } \neq \in \{<, \supset, =\}, i \in \{\text{protocol, src_ip, src_port, dst_ip, dst_port}\}$$

For example, rule 1 and rule 2 below are completely disjoint since all corresponding fields in both rules are distinct.

```
1: tcp, 140.192.37.10, 2000, 163.122.51.50, 80, accept
2: udp, 140.192.37.20, 3000, 163.122.51.60, 21, accept
```

Definition 4: Rules R_x and R_y are *partially disjoint* (or *partially matched*) if there is at least one field in R_x that is a subset or a superset or equal to the corresponding field in R_y , and there is at least one field in R_x that is not a subset and not a superset and not equal to the corresponding field in R_y . Formally:

R_x and R_y are *partially disjoint* (or *partially matched*) iff

$$\exists i, j \text{ such that: } R_x[i] \neq R_y[i] \quad \text{and} \quad R_x[j] \equiv R_y[j]$$

$$\text{where } \neq \in \{<, \supset, =\} \quad \text{and} \quad i, j \in \{\text{protocol, src_ip, src_port, dst_ip, dst_port}\}$$

For example, rule 1 and rule 2 below are partially disjoint (or partially matched) since all fields in rule 1 are related to the corresponding fields in rule 2 except the destination port field.

```
1: tcp, 140.192.37.10, any, *.*.*.*, 80, accept
2: tcp, 140.192.37.*, any, *.*.*.*, 21, deny
```

Definition 5 Rules R_x and R_y are *correlated* if some fields in R_x are subsets or equal to the corresponding fields in R_y , and the rest of the fields in R_x are supersets of the corresponding fields in R_y . Formally:

R_x and R_y are *correlated* iff

$$\forall i: R_x[i] \neq R_y[i] \quad \text{and}$$

$$\exists i, j \text{ such that: } R_x[i] \subset R_y[i] \quad \text{and} \quad R_x[j] \supset R_y[j]$$

$$\text{where } i, j \in \{\text{protocol, src_ip, src_port, dst_ip, dst_port}\}$$

For example, Rule 1 and rule 2 below are correlated since they have the same protocol, source and destination ports, and the source address of rule 1 is a subset of the corresponding fields in rule 2, and the destination address of rule 1 is a superset of that of rule 2.

```
1: tcp, 140.192.37.10, any, *.*.*.*, 80, accept
2: tcp, *.*.*.*, any, 140.192.37.*, 80, deny
```

The diagrams shown in Figure 1 illustrate these relations between two filtering rules, R_x and R_y . Below we give an intuition of our proof that there is no other relation between R_x and R_y could exist. However, the full proof can be found in [1].

Theorem. *The union of these relations, \mathfrak{R} , represents the universal set of relations between any two k -tuple filters in a firewall policy.*

Proof. Intuitively, we first prove that the relation between any two 2-tuple filters, R_x and R_y , must be in \mathfrak{R} . This is simply shown by enumerating permutations of all possible relations between the fields of R_x and R_y . Since any two fields can be related with any relation in $\{=, <, \supset, \neq\}$, thus there are 16 possible combinations for R_x and R_y relations (e.g., $R_x[\text{field}_1] = R_y[\text{field}_1]$ and $R_x[\text{field}_2] \supset R_y[\text{field}_2] \Rightarrow R_y \mathfrak{R}_{\text{IM}} R_x$

where \mathfrak{R}_{IM} represents inclusive match). This shows that any relation between R_x and R_y must be one of the rules relations in \mathfrak{R} . Next, we prove that adding one more field to any two filters related with one of the defined relations above will produce two filters, R_x' and R_y' , that are also related by one of the defined relations above. We can also show this by enumerating all combinations between R_x and R_y and the added field as follows. $R_x \mathfrak{R}_{IM} R_y$ and $R_x[\text{field}_{k+1}] \supset R_y[\text{field}_{k+1}] \Rightarrow R_y' \mathfrak{R}_C R_x'$ where \mathfrak{R}_{IM} and \mathfrak{R}_C represent inclusive match and correlated match relationships respectively.

Therefore, based on these two observations, we then prove by induction that any two rules with k -tuple filters must be related with each other by one of the rule relations defined in this section. The complete proof can be found in [1].

3.2. Firewall Rule Policy Representation

We represent the firewall rule policy by a single rooted tree that we name the *policy tree*. The tree model provides a simple and apprehensible representation of the filtering rules and at the same time allows for easy discovery of relations and anomalies among the rules. Each node in a policy tree represents a field of the filtering rule, and each branch at this node represents a possible value of the associated field. At every node, we use a hash table to store the field value for each emerging branch. The root node of a policy tree represents the protocol field, and the leaf nodes represent the action field, intermediate nodes represent other 5-tuple filter fields in order. Every tree path starting at the root and ending at a leaf represents a rule in the policy and vice versa. Rules that have the same field value at a specific node, will share the same branch representing that value.

Figure 3 illustrates the policy tree model of the security policy in Figure 2. Notice that every rule should have an action leaf in the tree. The dotted box below each leaf indicates the rule represented by that leaf. The tree shows that rules 1 to 9 share the same branch at the protocol node since they all have the same field value “tcp”. Rules 1 and 5 each has a separate source address branch as they have different field values. Rules 2, 4, 6 and 7 share the same source address branch as they all have the same field value “140.192.37.*”. Similarly, rules 3, 8 and 9 share the same source address branch. Notice that rule 8 has a separate branch, and also appears on rule 7 branch, because it is a superset of rule 7. Also notice that rule 4 has a separate branch, and also appears on rule 3 branch, as it is a subset of rule 3. Since rule 9 is a superset of all the previous rules, it should appear on the branches of these rules, but has been omitted for clarity.

order	protocol	src_ip	src_port	dst_ip	dst_port	action
1:	tcp,	140.192.37.20,	any,	*.*.*.*,	80,	deny
2:	tcp,	140.192.37.*,	any,	*.*.*.*,	80,	accept
3:	tcp,	*.*.*.*,	any,	140.192.37.40,	80,	accept
4:	tcp,	140.192.37.*,	any,	140.192.37.40,	80,	deny
5:	tcp,	140.192.37.30,	any,	*.*.*.*,	21,	deny
6:	tcp,	140.192.37.*,	any,	*.*.*.*,	21,	accept
7:	tcp,	140.192.37.*,	any,	140.192.37.40,	21,	accept
8:	tcp,	*.*.*.*,	any,	140.192.37.40,	21,	accept
9:	tcp,	*.*.*.*,	any,	*.*.*.*,	any,	deny
10:	udp,	140.192.37.*,	any,	*.*.*.*,	53,	accept
11:	udp,	*.*.*.*,	any,	140.192.37.*,	53,	accept
12:	udp,	*.*.*.*,	any,	*.*.*.*,	any,	deny

Figure 2. A firewall policy example.

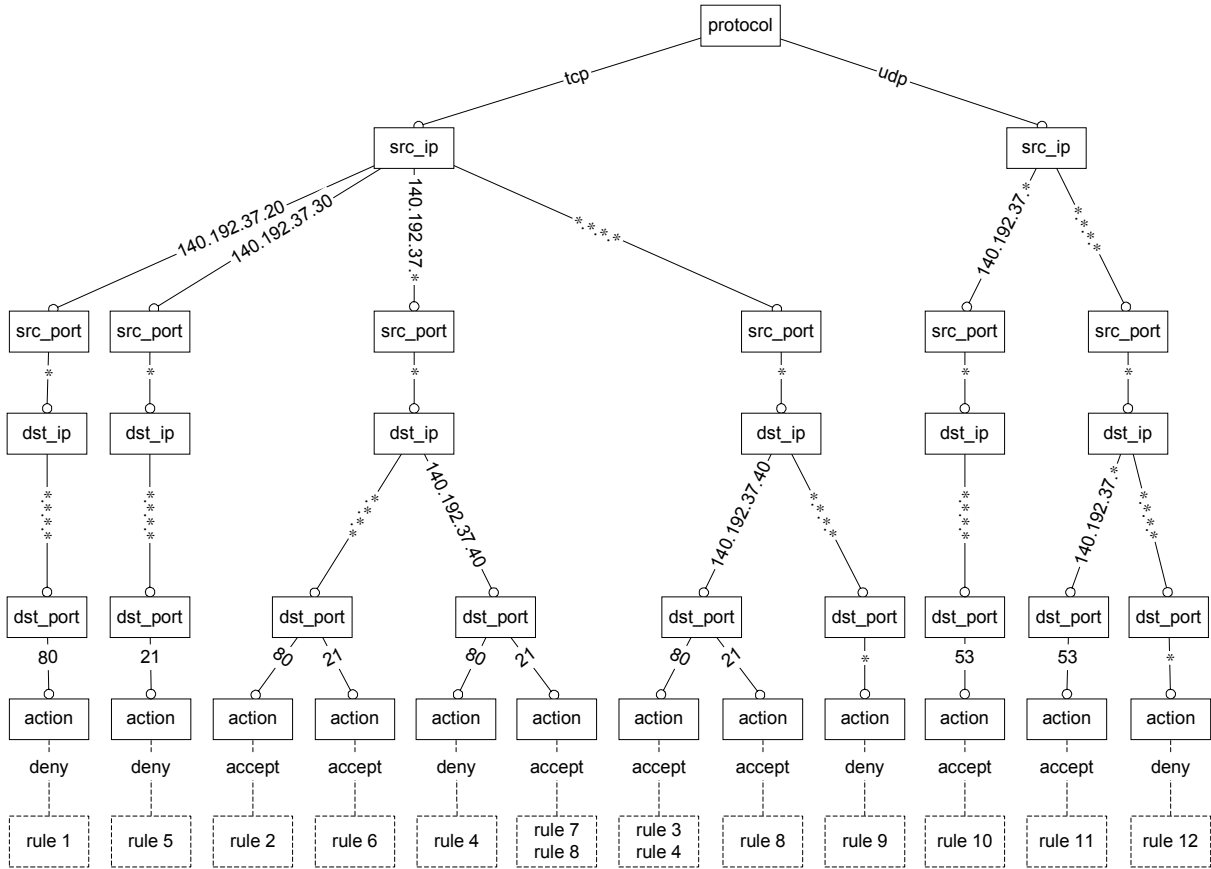


Figure 3. Policy tree for the firewall policy in Figure 2.

The algorithm shown in Figure 4 is used in building the policy tree. The basic idea is to insert the filtering rule in the correct tree path. While inserting a rule field at any tree node, the rule branch is determined based on matching the field value with the existing branches. If a branch exactly matches the rule, the rule is inserted in this branch; otherwise a new branch is created. The rule also propagates in superset or superset branches to preserve the relations between the policy rules. The algorithm is invoked for each rule in the policy with the rule as the first parameter, the protocol field as the second parameter and the root node as the third parameter. If the inserted field is not the rule action, the algorithm checks if the input field value matches any of the already created branches at the input node. If a match is found, the algorithm is recursively called with the same rule and the next field and the matching branch node as parameters. The recursive call ensures that a rule will propagate in all the branches representing rules that may be related to the newly inserted rule. If the field value was not equal to any of the existing branches, a new branch that represents the inserted field value is created. If the inserted field is not the action field, the algorithm is recursively applied on the new branch with the same rule and the next field and the new branch node as parameters.

We would like to emphasize that our main objective is to provide a simple model that can be used for the analysis of filtering rules, as well as visualizing the rules in an expressible view that will reflect relations between rules. The main objective of the “Firewall Policy Advisor” is to assist in firewall policy editing. Therefore, we are not as much concerned about search time complexity as we are concerned about clarity and simplicity of the rule model.

```

function BuildPolicyTree(rule, field, node)
  value_found = FALSE
  if field ≠ ACTION then
    for each branch in node.branch_list do
      if branch.value = rule.field.value then (* exact match found *)
        value_found = TRUE
        BuildPolicyTree(rule, field.next, branch.node)
      else if branch.value ⊂ rule.field.value (* inclusive match found *)
        or branch.value ⊃ rule.field.value then
          BuildPolicyTree(rule, field.next, branch.node)
        end if
      end if
    end for
  end if
  if value_found = FALSE then (* field value is not in tree *)
    new_branch = new TreeBranch(rule, rule.field, rule.field.value)
    node.branch_list.add(new_branch)
    if field ≠ ACTION then
      BuildPolicyTree(rule, field.next, new_branch.node)
    end if
  end if
end function

```

Figure 4. Algorithm for building the policy tree.

4. Firewall Policy Anomaly Detection

The ordering of filtering rules in a security policy is very crucial in determining the firewall policy because the firewall packet filtering process is performed by sequentially matching the packet against filtering rules until a match is found. If filtering rules are independent (or completely disjoint), the ordering of the rules is insignificant. However, it is very common to have filtering rules that are inter-related. In this case, if the relative rule ordering is not carefully assigned, some rules may be always screened by other rules producing an incorrect security policy and action. Moreover, when large number of filtering rules exists in a policy, the possibility of writing conflicting or redundant rules is relatively high. A firewall policy anomaly is defined as the existence if two or more different filtering rules that match the same packet. In this section, we define different types of anomalies that may exist among filtering rules in a firewall policy and then describe the technique to discover these anomalies.

4.1. Firewall Policy Anomaly Types

Here, we describe and then define a number of possible firewall policy anomalies. This includes clear conflicts that cause some rules to be always suppressed by other rules, or warnings for potential conflicts between related rules.

(1) Shadowing anomaly: A rule is shadowed when a previous rule matches *all* the packets that match this rule, such that the shadowed rule will never be evaluated. If the shadowed rule is removed, the security policy will not be affected. Rule R_x is shadowed by rule R_y if R_x follows R_y in the order, and R_x is a subset match of R_y , and the actions of R_x and R_y are different. As illustrated in the rules in Figure 2, rule 4 is a special case of rule 3 with a different action, so if rule 4 is removed, the effect of the resulting policy will be unchanged. We say that rule 4 is shadowed by rule 3.

Shadowing is a critical error in the policy, as the filtering rule never takes effect. This might cause a permitted traffic to be blocked and vice versa. It is important to discover shadowed rules and alert the administrator who might correct this error by reordering or removing the shadowed rule.

(2) Correlation anomaly: Two rules are correlated if the first rule in order matches *some* packets that match the second rule and the second rule matches *some* packets that match the first rule. Rule R_x and rule R_y have a correlation anomaly if R_x and R_y are correlated and the actions of R_x and R_y are different. As illustrated in the rules in Figure 2, rule 2 is in correlation with rule 3; if the order of the two rules is reversed, the effect of the resulting policy will be changed, but rule 2 will not be shadowed by rule 3.

Correlation is considered an anomaly warning because the correlated rules imply an action that is not explicitly handled by the filtering rules. Consider the following rules:

```
1: tcp, 140.192.37.10, any, *.*.*.*, 80, accept
2: tcp, *.*.*.*, any, 140.192.37.*, 80, deny
```

These two rules with this ordering imply that all HTTP traffic coming from address “140.192.37.10” and going to address “140.192.37.*” is accepted. However, if the order is reversed, the same traffic will be denied. Therefore, in order to resolve this conflict, we point out the correlations between the filtering rules and prompt the user to choose the proper order that complies with the security policy.

(3) Redundancy anomaly: A redundant rule performs the same action on the same packets as another rule such that if the redundant rule is removed, the security policy will not be affected. Rule R_x is redundant to rule R_y if R_x is a subset match of R_y and the actions of R_x and R_y are similar. As illustrated in the rules in Figure 2, rule 7 is redundant to rule 8, so if rule 7 is removed, the effect of the resulting policy will be unchanged.

Redundancy is considered an error. A redundant rule may not contribute in making the filtering decision, however, it adds to the size of the filtering rule table, and might increase the search time and space requirements. It is important to discover redundant rules so that the administrator may modify its filtering action or remove it altogether.

(4) Generalization anomaly: A rule is a generalization of another rule if the first rule matches *all* the packets that the second one could match but not the opposite. Rule R_x is a generalization of rule R_y if R_x follows R_y in the order, and R_x is a superset match of R_y and the actions of R_x and R_y are different. As illustrated in the rules in Figure 2, rule 2 is a generalization of rule 1; if the order of the two rules is reversed, the effect of the resulting policy will be changed, and rule 1 will not be effective anymore, as it will be shadowed by rule 2. Therefore, as a general guideline, if there is an inclusive match relationship between two rules, the superset (or general) rule should come after the subset (or specific) rule.

Generalization is considered only an anomaly warning because inserting a specific rule makes an exception of the general rule, and thus confirming this action by the administrator is important.

4.2. Anomaly Detection Algorithm

The state diagram shown in Figure 5 summarizes anomaly discovery for any two rules, R_x and R_y , where R_x comes after R_y in order. For simplicity, the source address and source port and integrated into one field, and the same with the destination address and port. This simplification reduces the number of states and simplifies the explanation of the diagram. A similar state diagram can be produced for the real case of five fields with a substantially larger number of states involved.

Initially no relationship is assumed. Each field in R_x is compared to the corresponding field in R_y , starting with the protocol then source address and port, and finally destination address and port. The relationship between the two rules is determined based on the result of subsequent comparisons. If every field of R_x is a subset or equal to the corresponding field in R_y and both rules have the same action, R_x will be redundant to R_y , while if the actions are different, R_x will be shadowed by R_y . These cases are represented by the state sequences: 0-1-4-8-11, 0-1-4-8-12, 0-1-5-8-11, 0-1-5-8-12, 0-2-5-8-11 and 0-2-5-8-12. If every field of R_x is a superset or equal to the corresponding field in R_y and both rules have the same action, R_y will be redundant to R_x , while if the actions are different, R_x is a generalization of R_y . These cases are represented by the state sequences: 0-1-6-9-12, 0-1-6-9-13, 0-1-4-6-9-12, 0-1-4-6-9-13, 0-0-3-6-9-12 and 0-0-3-6-9-13.

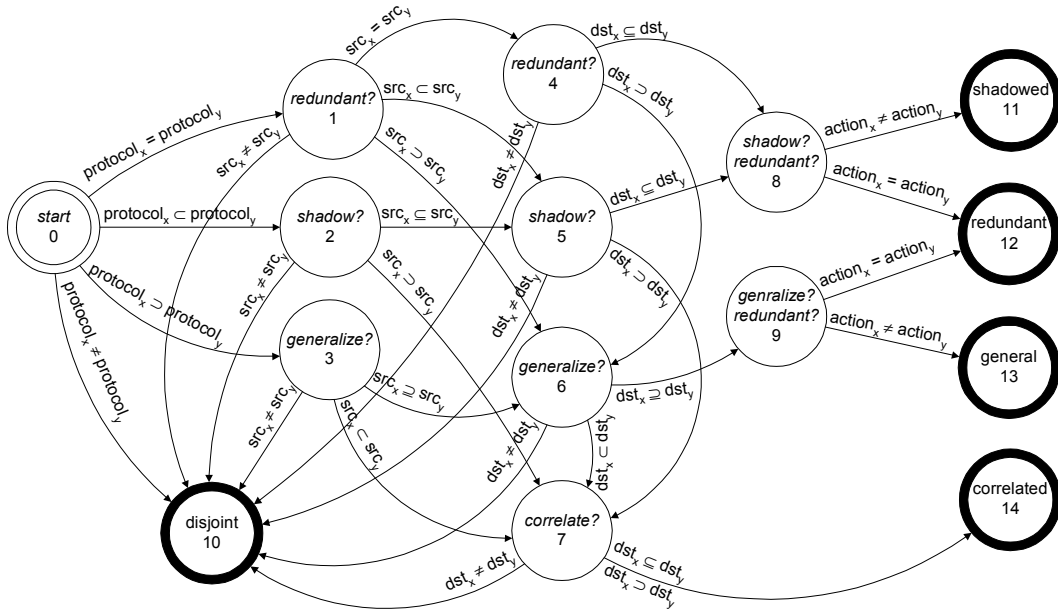


Figure 5. State diagram for detecting anomalies for rules R_x and R_y , rule R_x comes after R_y .

If some fields of R_x are subsets or equal to the corresponding fields in R_y , and some fields of R_x are supersets to the corresponding fields in R_y , and their actions are different, then R_x is in correlation with R_y . This case is represented by the state sequences: 0-1-5-7-14, 0-1-6-7-14, 0-1-4-6-7-14, 0-2-7-14, 0-2-5-7-14, 0-3-7-14 and 0-3-6-7-14. If any field of R_x is not a subset, not a superset and not equal to the corresponding field in R_y , then R_x and R_y are disjoint. This case is represented by the state sequences: 0-10, 0-1-10, 0-2-10, 0-3-10, 0-1-4-10, 0-1-5-10, 0-1-6-10, 0-2-5-10, 0-2-7-10, 0-3-6-10 and 0-3-7-10.

The basic idea for discovering anomalies is by determining if two rules coincide in their policy tree paths. If the tree path of a rule coincides with the tree path of another rule, there is a potential anomaly that can be determined based on the previous definitions of anomalies. If rule paths do not coincide, these rules are disjoint and they have no anomalies. The algorithm for building the policy tree is extended to determine the relationships and anomalies among the filtering rules. An extra algorithm parameter is added to save the anomaly state. The extended algorithm is shown in Figures 6 and 7. The algorithm is divided into two main parts: an *anomaly discovery routine*, which represents the transition states in the state diagram, and an *anomaly decision routine*, which represents the termination states.

In the discovery routine, the previous anomaly state is checked if there is a value match between the field of the new rule and the already existing field branch. The next anomaly state is determined based on the shown state diagram and the algorithm is executed recursively to let the rule propagate in existing branches and check the remaining fields. As the rule propagates, the anomaly state is updated until the final state is reached. If a field value exactly matches an existing branch, a potential redundancy state is recorded. If the field is a subset of the branch, a potential shadowing state is recorded except if the previous state is generalization; in this case the anomaly state is changed to correlation. If the field is a superset of the branch, a potential generalization state is recorded except if the previous state is shadowing; in this case the anomaly state is changed to correlation. If there is no match for the value of a field, a new branch is created at the current node to represent the inserted field value, and the anomaly state is initialized to no anomaly. The algorithm then proceeds to check the next field.

```

function DiscoverAnomaly(rule, field, node, anomaly_state)
  if field  $\neq$  ACTION then
    value_found = FALSE
    for each branch in node.branch_list do
      if branch.value = rule.field.value then
        value_found = TRUE
        if anomaly_state = NOANOMALY then
          anomaly_state = REDUNDANT
        DiscoverAnomaly(rule, field.next, branch.node, anomaly_state)
      else
        if rule.field.value  $\subset$  branch.value then
          if anomaly_state = GENERALIZATION then
            DiscoverAnomaly(rule, field.next, branch.node, CORRELATION)
          else
            DiscoverAnomaly(rule, field.next, branch.node, SHADOWING)
        else if rule.field.value  $\supset$  branch.value then
          if anomaly_state = SHADOWING then
            DiscoverAnomaly(rule, field.next, branch.node CORRELATION)
          else
            DiscoverAnomaly(rule, field.next, branch.node GENERALIZATION)
          end if
        end if
      end for
    if value_found = FALSE then
      new_branch = new TreeBranch(rule, rule.field, rule.field.value);
      node.branch_list.add(new_branch);
      DiscoverAnomaly(rule, field.next, new_branch.node, NOANOMALY);
    end if
  else /* action field reached */
    call DecideAnomaly(rule, field, node, anomaly_state)
  end if
end function

```

Figure 6. Algorithm for building the policy tree with anomaly discovery.

The decision routine is activated once all the rule fields have been inserted in the tree and the action field is reached. At that point the final anomaly state is determined and reported. If the rule action coincides with the action of another rule, an anomaly is discovered. Correlation and generalization are confirmed if the rule actions are different. If the input anomaly state is a generalization and the actions are the same, the existing rule is redundant to the new rule. Finally, if the new rule is a subset or equal to the existing rule, the new rule is redundant if their actions are the same, and is shadowed if their actions are different. If an anomaly is discovered and decided, the user is reported with the type of anomaly and the rules involved. The anomaly is also recorded in the filtering rule list.

It is important to mention that we consider this algorithm as an offline process that precedes the actual deployment of the filtering rules in the firewall rule table. As the “Firewall Policy Advisor” is a design-time tool, our main focus is on correctness and usability more than the computation complexity and optimization of the algorithm.

Applying the algorithm on the rules in Figure 2, two conflicts are detected. Rule 4 is shadowed by rule 3 since they both coincide in the tree path, and they have different filtering actions. Rule 7 is redundant since rule 8 propagates in its tree path and has a similar action.

Figure 8 shows the graphical user interface for the “Firewall Policy Advisor.” The bottom panel shows a tabular list of filtering rules. The top-left panel displays the policy tree showing aggregated rules. The top-right panel displays the anomalies discovered along with highlighting redundant and shadowed rules in a different color.

```

function DecideAnomaly(rule, field, node, anomaly)
  if node has branch_list then
    branch = node.branch_list.first()
    if anomaly = CORRELATION then
      if rule.action ≠ branch.value then
        report rule rule.id is in correlation with rule branch.rule.id
      else if anomaly = GENERALIZATION and rule.action ≠ branch.value then
        report rule rule.id is a generalization of rule branch.rule.id
      else if anomaly = GENERALIZATION and rule.action = branch.value then
        branch.rule.setAnomaly(REDUNDANCY)
        report rule branch.rule.id is redundant to rule rule.id
      else if rule.action = branch.value then
        anomaly = REDUNDANCY
        report rule rule.id is redundant to rule branch.rule.id
      else if rule.action ≠ branch.value then
        anomaly = SHADOWING
        report rule rule.id is shadowed by rule branch.rule.id
      end if
    end if
  end if
  rule.setAnomaly(anomaly)
end function

```

Figure 7. Algorithm for making the anomaly decision.

The screenshot shows the Policy Advisor application window. It features three main panels: a Policy Tree on the left, an Anomalies list in the middle, and a Rule List table at the bottom. The Policy Tree shows a hierarchy of nodes including 'pt tcp', 'sa 140.192.37.20/32', 'sa 140.192.37.0/8', 'sp 0', 'da 140.192.37.40/32', 'dp 80', 'dp 21', and 'sa 0.0.0.0/0'. The Anomalies list contains 11 entries describing relationships between rules, such as 'rule 2 is a generalization of rule 1' and 'rule 3 is in correlation with rule 1'. The Rule List table below contains 9 rows of rule data.

Rule	Protocol	Source IP	Source Port	Destin IP	Destin Port	Action
1	tcp	140.192.37.20/32	0	0.0.0.0/0	80	deny
2	tcp	140.192.37.0/8	0	0.0.0.0/0	80	accept
3	tcp	0.0.0.0/0	0	140.192.37.40/32	80	accept
4	tcp	140.192.37.0/8	0	140.192.37.40/32	80	deny
5	tcp	140.192.37.30/32	0	0.0.0.0/0	21	deny
6	tcp	140.192.37.0/8	0	0.0.0.0/0	21	accept
7	tcp	140.192.37.0/8	0	140.192.37.40/32	21	accept
8	tcp	0.0.0.0/0	0	140.192.37.40/32	21	accept
9	tcp	0.0.0.0/0	0	0.0.0.0/0	0	deny

Figure 8. Policy Advisor anomaly detection user interface.

5. Firewall Policy Editor

Firewall policies are often written by different network administrators and occasionally updated (inserting, modifying or removing rules) to accommodate new security requirements and network topology changes. Editing a security policy for inserting, removing or modifying rules can be far more difficult than creating a new one. As rules in firewall policy are often ordered, a new rule must be inserted in a particular order in order to avoid creating anomalies. The same applies if any network field in a rule is modified. In this section, we present a policy editor tool that simplifies the rule editing task significantly, and avoids introducing anomalies due to policy updates. The policy editor (1) prompts the user with the proper position(s) for a new or modified rule, (2) shows the changes in the security policy semantic before and after removing a rule, and (3) provides visual aids for users to track and verify policy changes. Using the policy editor, administrators require no prior knowledge or understating of the firewall policy in order to insert, modify or remove a rule. In the following, we present the editing, modifying and removal process as implemented in our policy editor.

5.1. Rule Insertion

Since the ordering of rules in the filtering rule list directly impacts the semantics of the firewall security policy, a new rule must be inserted in the proper order in the policy such that no shadowing, correlation or redundancy is created. The policy editor helps the user to determine the correct position(s) of the new rule to be inserted. It also identifies anomalies that may occur due to improper insertion of the new rule, and suggests the proper resolution.

The algorithm shown in Figure 9 describes the mechanism to insert a new rule. The general idea is that the order of a new rule is determined based on its relation with other existing rules in the firewall policy. In general, a new rule should be inserted before any rule that is its superset match, and after any rule that is its subset match. The policy tree is used to keep track of the correct order of the new rule, and detect any potential anomalies. The algorithm is organized into two phases: the *browsing phase* and the *insertion phase*. In the browsing phase, the fields of the new rule are compared with the corresponding tree node values one at a time. If the field value of the new rule is a subset of an existing branch, then the new rule must be inserted *before* the minimum order of all the rules/leaves in this branch. If the field value is a superset of an existing branch, the rule must be inserted *after* the maximum order of all the rules in this branch. In addition, if the field value is an *exact match* or a *subset match* of a branch, evaluating the next field continues recursively by browsing through the branch sub-tree until correct position of the rule within the sub-tree is determined. Otherwise, if *disjoint* or *superset* match occurs, a branch is created for the new rule.

The algorithm enters into the insertion phase when the action field of a new rule is to be inserted. If an action branch is created for the new rule, then the rule will be inserted and assigned the order determined in the browsing phase. If there is more than one possible order for this rule, the user is asked to select an order from within a valid range of orders as determined in the browsing phase. However, if the order state of the new rule remains UNDETERMINED or it coincides with the branches for all 5-tuple fields of an existing rule, which has the same action, then policy editor ignores this new rule and prompts the user with the appropriate message. In the former case, the rule exactly matches an existing rule and considered redundant or directly conflicting depending on the action. In the latter case, the new rule is an inclusive subset match of an existing rule but they have the same action, and thus it is considered redundant rule. If the rule is inserted, the anomaly detection algorithm is invoked to alert the administrators with any generalization or correlation cases as a possible source of anomalies the firewall policy.

```

function InsertRule(rule, field, node)
  read rule.field.value
  if field ≠ ACTION then
    for each branch in node.branch_list do
      if branch.value = rule.field.value then
        target_branch = branch
      else
        if rule.field.value ⊂ branch.value then
          if rule.max_order > branch.rule_list.getMinOrder() then
            rule.max_order = branch.rule_list.getMinOrder()
            target_branch = branch
          end if
        else if rule.field.value ⊃ branch.value then
          if rule.min_order < branch.rule_list.getMaxOrder() then
            rule.min_order = branch.rule_list.getMaxOrder()+1
          end if
        end if
      end if
    end for
    if exists target_branch then
      InsertRule(rule, field.next, target_branch.node)
    else
      new_branch = new TreeBranch(rule, rule.field, rule.field.value);
      node.branch_list.add(new_branch);
      InsertRule(rule, field.next, new_branch.node);
    end if
  else
    read rule.order where rule.max_order ≥ rule.order ≥ rule.min_order
    if node has branch_list then
      if rule.min_order=UNDERTERMINED and rule.max_order=UNDERTERMINED then
        branch = node.branch_list.first()
        if rule.action = branch.value then
          error rule rule.id is redundant to rule branch.rule.id
        else if branch.value ≠ rule.action then
          error rule rule.id is shadowed by rule branch.rule.id
        end if
      else if rule.max_order ≠ UNDERTERMINED and
        rule.action = branch.value then
          error rule rule.id is redundant to rule branch.rule.id
        end if
      end if
      rule_list.insertAt(rule, rule.order)
      DiscoverAnomaly(rule, tree_root, anomaly)
      if anomaly in {CORRELATION, GENERALIZATION} then
        alert potential anomaly, please confirm policy semantics
      end if
    end if
  end function

```

Browse phase

Insertion phase

Figure 9. Algorithm for inserting a new rule in the policy.

5.2. Rule Removal and Modification

In general, removing a rule has much less impact on the firewall policy than insertion. A removed rule does not introduce an anomaly but it might change the policy semantics and this change should be highlighted and confirmed. To remove a rule, the user enters the rule number to retrieve the rule from the rule list and selects to remove it. To preview the effect of rule removal, the policy editor gives a textual translation of the affected portion of the policy before and after the rule is removed. The user is able to compare and inspect the policy semantics before and after removal, and re-assure correctness of the policy changes. Modifying a rule in a firewall policy is also a critical operation. However, this editing action can be easily managed as rule removal and insertion as described before.

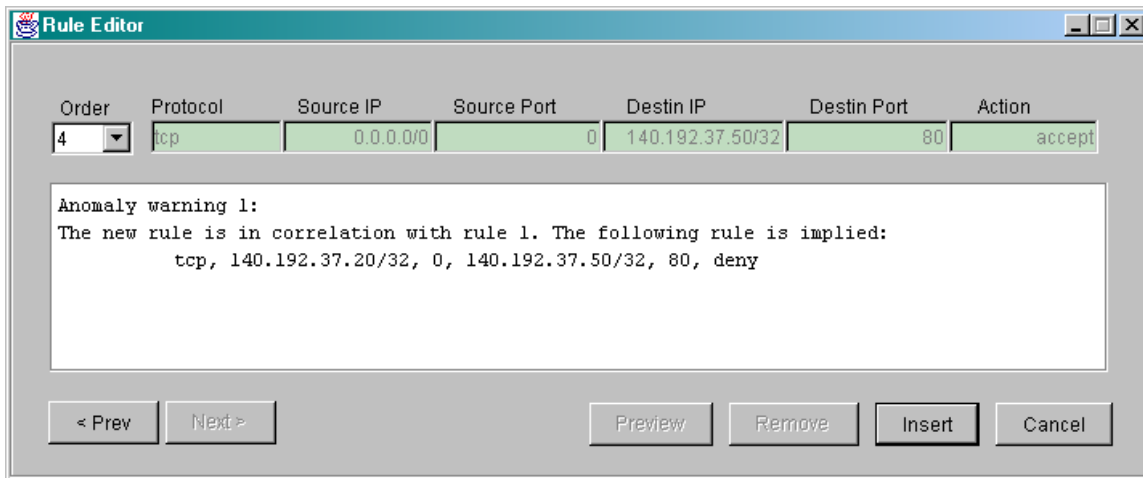


Figure 10. Rule editor user interface.

Figure 10 shows the graphical user interface for the rule editor tool. The figure shows the final step in inserting a rule in the filtering rule table. The tool alerts the user for any anomalies that may be introduced by inserting the new rule.

6. Firewall Policy Translator

It is difficult to comprehend the policy semantics by reading through a set of filtering rules. Rules that have common or related fields values such as source IP address or destination port number may be widely scattered within the rule table, making it even harder to have a concise and complete description of the firewall policy. In this part of our work, we implemented a firewall policy translation tool that describes, in a natural textual language, the meaning and the interactions of all filtering rules in the policy, revealing the complete semantics of the policy in a very concise fashion. The produced policy translation should have the following features:

- **Complete:** the translator should reflect each and every correct rule in the firewall rule table and preserve the relations between related rules.
- **Concise:** the translator should express the firewall policy in the shortest text possible. For this purpose, the policy tree is represented in many different field ordering formats in order to aggregate as many related rules as possible in a single translated statement.
- **Easy to read:** the translation text and fields should be properly formatted and aligned, respectively, to be easily comprehensible.

In this section, we describe the design and the implementation of the policy translator, which is based on the policy tree model presented in Section 3.

6.1. Maximum Rule Aggregation for Tree-based Translation

To achieve a concise translation, we model the rules such that partially disjoint rules are aggregated together based on related field values. Let us start by translating rule 2 and rule 6 from Figure 2 separately, we get the following translation: “*accept tcp traffic from address 140.192.37.* and to port 80*” for rule 2 and “*accept tcp traffic from address 140.192.37.* and to port 21*” for rule 6. Both rules have common protocol, source address, source port and destination address but different destination ports. The policy tree aggregates the two rules in one branch starting from the protocol node down to the destination address node, and then the two rules split in two branches at the destination port node. By using depth-first traversal to translate the branch that aggregates the two rules, we get the following concise translation that exploits the commonality of the two rules: “*accept tcp traffic from address 140.192.37.* and to port 80 or to port 21.*”


```

2:  accept all tcp traffic from address 140.192.37.* and { to port 80
6:                                     or to port 21 }
1:          except from address 140.192.37.20 and to port 80
5:          or from address 140.192.37.30 and to port 21
3:  accept all tcp traffic to address 140.192.37.40 and { to port 80
8:                                     or to port 21 }
9:  deny any other tcp traffic
10: accept all udp traffic to port 53 and { from address 140.192.37.*
11:                                       or to address 140.192.37.* }
12: deny any other udp traffic

```

Figure 12. Translation of Figure 2 rules using depth-first traversal of the translation tree in Figure 11.

6.2. Translation Tree Construction Algorithm

The algorithm shown in Figure 13 is used to build the translation tree. Before using the translation algorithm, the anomaly detection algorithm is run to determine and exclude any shadowed or redundant rules. The root node of the translation tree is then created to represent the protocol field, and all non-conflicting rules are stored in the root node. The algorithm is then called with the root node as the first parameter, and an initial field order as the second parameter. The initial field order is arbitrary and will be optimized by the algorithm.

The first stage of the algorithm creates a list of policy trees each of which is rooted by a different rule field. The algorithm first excludes the field represented by the current node from the input field order. If there are no more fields to insert, the algorithm terminates after inserting the leaf action node. If there are more fields to insert then for every listed field a policy tree rooted by this field is created and added to a tree list. After this stage a list of policy trees, each rooted by a different field will be available. The branches emerging from the root of each tree are then added to a branches list, which will eventually contain all the branches of all the created policy trees.

In the second stage, the algorithm extracts the branches that have the maximum aggregation of rules from the branches list. The branches list is first sorted based on the number of rules aggregated in each branch. The branch having the maximum rule aggregation is extracted from the branches list and a new node representing the branch root field is inserted in the tree. After a branch is extracted, all the rules aggregated by this branch are removed from the rest of the branches. The second stage is repeated until all policy rules are aggregated in the extracted branches and there are no more uncovered rules. After this stage a single rooted sub-tree is created. Each branch at the root node represents a field value and aggregates the maximum number of rules that have this field value in common. It is important to notice that the sub-tree root may represent more than one field since the root branches may come from more than one policy tree, each with a different root field.

The last stage recursively calls the algorithm for each branch of the resulting sub-tree. This stage assures that at each level in the translation tree, all sub-tree branches are the optimal aggregate of the rules they represent. Translation of the rules is as simple as running a depth-first tree traversal algorithm on the translation tree. The traversal algorithm is extended to print prepositions and conjunctions between the printed field values to make the output translation as readable as possible.

Figure 14 shows the graphical user interface for the translation of the firewall policy given in Figure 2. Field conditions are hyperlinked with the rule table on the main window such that the user may know which rule is associated with each part of the translation.

```

function BuildTranslationTree(node, field_order)
  tree_order = field_order - node.field_id
  if tree_order.isEmpty() then
    node.add(new RuleTreeNode(node.rule(), ACTION, node.rule().action))
    return
  end if
  for each root_field in tree_order do
    tree = new PolicyTree(root_field, node.rules)
    tree.buildTree()
    tree_list.add(tree)
  end for
  for each tree in tree_list
    branch_list.add(tree.root)
  end for
  all_rules = node.rules
  while not all_rules.isEmpty() do
    sort(branch_list)
    branch = branch_list.firstElement()
    branch_list.remove(branch)
    child = new TreeNode(branch.rules, branch.field_id, branch.value)
    node.add(child)
    for each other_branch in branch_list do
      for each rule in branch do
        other_branch.removeRule(rule)
        all_rules.removeRule(rule)
      end for
    end for
  end while
  for each child in node.children do
    BuildTranslationTree(child, field_order)
  end for

```

Figure 13. Algorithm for building the translation tree.

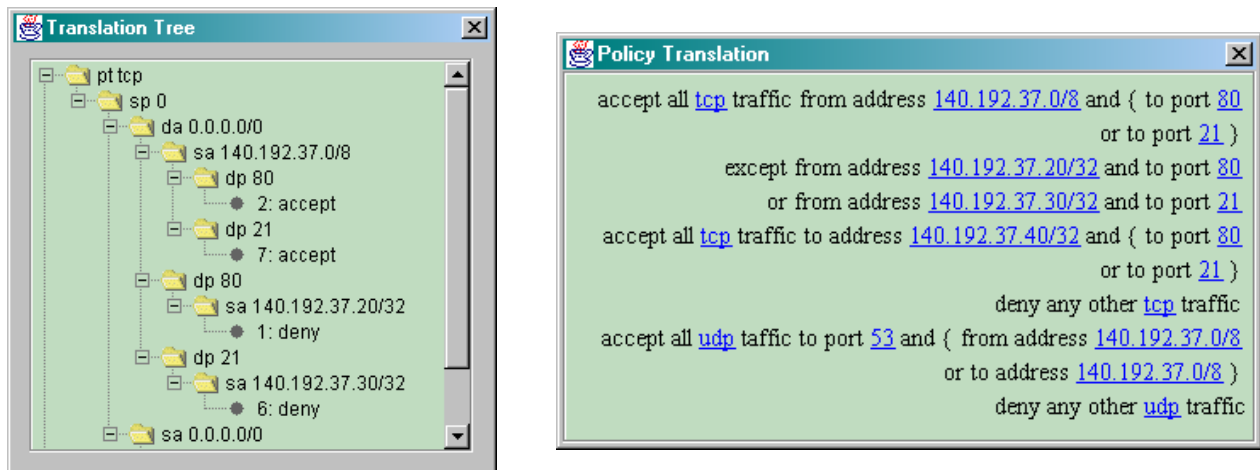


Figure 14. Translation tree and policy translation user interface.

7. Related Work

There is a significant amount of work has been reported in the area of firewall and policy-based security management. In this section, we focus our study on related work that intersects with our work in three areas: filtering modeling, conflict detection and rules analysis.

Several models have been proposed for filtering rules. Ordered binary decision diagram is used as a model for optimizing packet classification in [11]. Another model using tuple space is developed in [16], which combines a set of filters in one tuple and stored in a hash table. The model in [17] uses bucket filters indexed by search trees. Multi-dimensional binary tries are also used to model filters [15]. In [6] a geometric model is used to represent 2-tuple filtering rules. Because these models were designed particularly to optimize packet classification in high-speed networks, we found them too complex to use for firewall policy rule analysis such as anomaly detection, translation and editing. We can confirm from experience that the tree-based model is simple and powerful enough for this purpose.

Research in policy conflict analysis has been actively growing for many years. However, most of the work in this area addresses general management policies rather than firewall-specific policies. For example, authors in [13] classify possible policy conflicts in role-based management frameworks, and develop techniques to discover them. A policy conflict scheme for IPSec is presented in [8]. Although this work is very useful as a general background, it is not directly applicable in firewall anomaly detection. On the other hand, few research projects address the conflict problem in filtering rules. Both [6] and [10] provide algorithms for detecting and resolving conflicts among general packet filters. However, they only detect what we defined as correlation anomaly because it causes ambiguity in packet classifiers.

A considerable amount of research work has been performed in translating high-level firewall policy requirements to low-level filtering rules. The first generation of global policy management technology is presented in [9], which proposes a global policy definition language along with algorithms for verifying the policy and generating filtering rules. However, in [2] the authors adopted a better approach by using a modular architecture that separates the security policy and the underlying network topology to allow for flexible modification of the network topology without the need to update the security policy. Similar work has been done in [12] with a procedural policy definition language. Other research work go one step forward by offering query-based tools for firewall policy analysis. In [14] and [18], the authors developed a firewall analysis tool to perform queries on a set of filtering rules and extract the related rules in the policy. However, the extracted information is not expressive because it is represented in a low-level format (rules). In [7], an expert system is used for verifying the functionality of filtering rules by performing queries. In conclusion, we could not find any published research work that uses low-level filtering rules to perform a complete anomaly analysis, and provides editing, textual translation and visualization support for firewall policies.

8. Conclusions and Future Work

Firewall security, like any other technology, requires proper management to provide proper security service. Thus, just having a firewall on the boundary of a network may not necessarily make the network any secure. One reason of this is the complexity of managing firewalls rules and the potential network vulnerability due to rule conflicts (e.g. shadowing, correlation). The “Firewall Policy Advisor” presented in this paper provides a number of user-friendly tools for purifying and protecting the firewall policy from anomalies. The administrator can use the firewall policy advisor to manage a general firewall security policy without prior analysis of filtering rules. In this work, we formally defined all possible firewall rule relations and we used this to classify firewall policy anomalies. We then model the firewall rule information and relations in a tree-based representation. Based on this model and formalization, the firewall policy advisor implements three management tools:

- *Policy Anomaly Detector* for identifying conflicting, shadowing, correlated and redundant rules. When a rule anomaly is detected, users are prompted with proper corrective actions. We intentionally made the tool not to automatically correct the discovered anomaly but rather alarm the user because we believe that the administrator is the one who should do the policy changes.
- *Policy Editor* for facilitating rules insertion, modification and deletion. The policy editor automatically determines the proper order for any inserted or modified rule. It also gives a preview of the changed parts of the policy whenever a rule is removed to show the affect on the policy before and after the removal.

- *Policy Translator* for visualizing the rules and provide a concise natural language translation of any low-level filtering rules in a firewall policy. A considerable policy tree analysis is performed in order to combine the maximum aggregate of fields in one tree and achieve the most concise and precise translation.

The firewall policy advisor is shown to be very useful and effective when used on real firewall rules in different academic and industrial environments [1]. However, we believe that there is more to do in firewall policy management area. Our future research plan includes extending the translator with a query-based interface, extending the proposed techniques to handle distributed firewall policies with centralized or distributed repositories, classifying different semantics in firewall policies and extracting them from the filtering rules, enhancing our visualization of firewall policy rules.

Acknowledgements

We gratefully thank Iyad Kanj for his feedback on the theory work in this paper. We would also like to thank Lopamudra Roychoudhuri and Yongning Tang for their useful comments on an earlier version of this paper.

References

- [1] E. Al-Shaer and H. Hamed. "Design and Implementation of Firewall Policy Advisor Tools." *Technical Report, CTI-techrep0801*, School of Computer Science Telecommunications and Information Systems, DePaul University, August 2002.
- [2] Y. Bartal., A. Mayer, K. Nissim and A. Wool. "Firmato: A Novel Firewall Management Toolkit." In *Proceedings of 1999 IEEE Symposium on Security and Privacy*, May 1999.
- [3] D. Chapman and E. Zwicky. *Building Internet Firewalls, Second Edition*. Orielly & Associates Inc., 2000.
- [4] W. Cheswick and S. Belovin. *Firewalls and Internet Security*. Addison-Wesley, 1995.
- [5] S. Cobb. "ICSA Firewall Policy Guide v2.0." In *NCSA Security White Paper Series*, 1997.
- [6] D. Eppstein and S. Muthukrishnan. "Internet Packet Filter Management and Rectangle Geometry." In *Proceedings of 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2001.
- [7] P. Eronen and J. Zitting. "An Expert System for Analyzing Firewall Rules." In *Proceedings of 6th Nordic Workshop on Secure IT-Systems (NordSec 2001)*, November 2001.
- [8] Z. Fu, F. Wu, h. Huang, K. Loh, F. Gong, I. Baldine and C. Xu. "IPSec/VPN Security Policy: Correctness, Conflict Detection and Resolution." In *Proceedings of Policy'2001 Workshop*, January 2001.
- [9] J. Guttman. "Filtering Posture: Local Enforcement for Global Policies." In *Proceedings of 1997 IEEE Symposium on security and Privacy*, May 1997.
- [10] B. Hari, S. Suri and G. Parulkar. "Detecting and Resolving Packet Filter Conflicts." In *Proceedings of IEEE INFOCOM'00*, March 2000.
- [11] S. Hazelhusrt. "Algorithms for Analyzing Firewall and Router Access Lists." In *Technical Report TR-WitsCS-1999*, Department of Computer Science, University of the Witwatersrand, South Africa, July 1999.
- [12] S. Hinrichs. "Policy-Based Management: Bridging the Gap." In *Proceedings of 15th Annual Computer Security Applications Conference (ACSAC'99)*, December 1999.
- [13] E. Lupu and M. Sloman. "Conflict Analysis for Management Policies." In *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM'1997)*, May 1997.
- [14] A. Mayer, A. Wool and E. Ziskind. "Fang: A Firewall Analysis Engine." In *Proceedings of 2000 IEEE Symposium on Security and Privacy*, May 2000.
- [15] L. Qiu, G. Varghese, and S. Suri. "Fast Firewall Implementations for Software and Hardware-based Routers." In *Proceedings of 9th International Conference on Network Protocols (ICNP'2001)*, November 2001
- [16] V. Srinivasan, S. Suri and G. Varghese. "Packet Classification Using Tuple Space Search." In *Computer ACM SIGCOMM Communication Review*, October 1999.
- [17] T. Woo. "A Modular Approach to Packet Classification: Algorithms and Results." In *Proceedings of IEEE INFOCOM'00*, March 2000.
- [18] A. Wool. "Architecting the Lumeta Firewall Analyzer." In *Proceedings of 10th USENIX Security Symposium*, August 2001.
- [19] Cisco Secure Policy Manager 2.3 Data Sheet.
http://www.cisco.com/warp/public/cc/pd/sqsw/sqppmn/prodlit/spmgr_ds.pdf
- [20] Check Point Visual Policy Editor Data Sheet.
http://www.checkpoint.com/products/downloads/vpe_datasheet.pdf