Detection and Resolution of Anomalies in Firewall Policy Rules

Muhammad Abedin, Syeda Nessa, Latifur Khan, and Bhavani Thuraisingham

Department Of Computer Science
The University of Texas at Dallas
{maa056000,skn051000,lkhan,bxt043000}@utdallas.edu

Abstract. A firewall is a system acting as an interface of a network to one or more external networks. It implements the security policy of the network by deciding which packets to let through based on rules defined by the network administrator. Any error in defining the rules may compromise the system security by letting unwanted traffic pass or blocking desired traffic. Manual definition of rules often results in a set that contains conflicting, redundant or overshadowed rules, resulting in anomalies in the policy. Manually detecting and resolving these anomalies is a critical but tedious and error prone task. Existing research on this problem have been focused on the analysis and detection of the anomalies in firewall policy. Previous works define the possible relations between rules and also define anomalies in terms of the relations and present algorithms to detect the anomalies by analyzing the rules. In this paper, we discuss some necessary modifications to the existing definitions of the relations. We present a new algorithm that will simultaneously detect and resolve any anomaly present in the policy rules by necessary reorder and split operations to generate a new anomaly free rule set. We also present proof of correctness of the algorithm. Then we present an algorithm to merge rules where possible in order to reduce the number of rules and hence increase efficiency of the firewall.

Keywords. Packet Filters, Network Security, Firewalls, Anomalies, Security Policy.

1 Introduction

A firewall is a system that acts as an interface of a network to one or more external networks and regulates the network traffic passing through it. The firewall decides which packets to allow to go through or to drop based on a set of "rules" defined by the administrator. These rules have to be defined and maintained with utmost care, as any slight mistake in defining the rules may allow unwanted traffic to be able to enter or leave the network, or deny passage to quite legitimate traffic. Unfortunately, the process of manual definition of the rules and trying to detect mistakes in the rule set by inspection is very prone to errors and consumes a lot of time. Thus, research in the direction of detecting

anomalies in firewall rules have gained momentum of recent. Our work focuses on automating the process of detecting and resolving the anomalies in the rule set.

Firewall rules are usually in the form of a criteria and an action to take if any packet matches the criteria. Actions are usually *accept* and *reject*. A packet arriving at a firewall is tested with each rule sequentially. Whenever it matches with the criteria of a rule, the action specified in the rule is executed, and the rest of the rules are skipped. For this reason, firewall rules are order sensitive. When a packet matches with more than one rules, the first such rule is executed. Thus, if the set of packets matched by two rules are not disjoint, they will create anomalies. For instance, the set of packets matching a rule may be a superset of those matched by a subsequent rule. In this case, all the packets that the second rule could have matched will be matched and handled by the first one and the second rule will never be executed. More complicated anomalies may arise when the sets of packets matched by two rules are overlapped.

If no rule matches the packet, then the default action of the firewall is taken. Usually such packets are dropped silently so that nothing unwanted can enter or exit the network. In this paper, we assume that the default action of the firewall system is to *reject* and develop our algorithms accordingly.

Of recent, research work on detecting and resolving anomalies in firewall policy rules have gained momentum. Mayer et al. present tools for analyzing firewalls in [13]. In [8], Eronen et al. propose the approach of representing the rules as a knowledge base, and present a tool based on *Constraint Logic Programming* to allow the user to write higher level operations and queries. Works focusing on automating the process of detecting anomalies in policy include [12] where Hazelhurst describes an algorithm to represent the rules as a *Binary Decision Diagram* and presents a set of algorithms to analyze the rules. Eppstein et al. give an efficient algorithm for determining whether a rule set contains conflicts in [7]. Al-Shaer et al. define the possible relations between firewall rules in [1, 2, 4], and then define anomalies that can occur in a rule set in terms of these definitions. They also give an algorithm to detect these anomalies, and present policy advisor tools using these definitions and algorithm. They extend their work to distributed firewall systems in [3, 5]. A work that focuses on detecting and resolving anomalies in firewall policy rules is [11], where they propose a scheme for resolving conflicts by adding resolve filters. However, this algorithm requires the support of prioritized rules, which is not always available in firewalls. Also their treatment of the criterion values only as prefixes makes their work specific. In [9], Fu et al. define high level security requirements, and develop mechanisms to detect and resolve conflicts among IPSec policies. Golnabi et al. describe a Data Mining approach to the anomaly resolution in [10].

Majority of current research focus on the analysis and detection of anomalies in rules. Those that do address the resolution of anomalies require special features or provisions from the firewall, or focus on specific areas. In this paper, we base our work on the research of Al-Shaer et. al. in [1, 2, 3] whose analysis is applicable to all rule based firewalls in general. However, their work is lim-

ited to the detection of anomalies. We also show that one of their definitions is redundant, and the set of definitions do not cover all possibilities. In our work, we remove the redundant definition, and modify one definition to cover all the possible relations between rules. We also describe the anomalies in terms of the modified definitions. Then we present a set of algorithms to simultaneously detect and resolve these anomalies to produce an anomaly-free rule set. We also present an algorithm to merge rules whenever possible. Reports are also produced by the algorithms describing the anomalies that were found, how they were resolved and which rules were merged.

The organization of the paper is as follows. In Sect. 2, we discuss the basic concepts of firewall systems, representation of rules in firewalls, possible relations between rules, and possible anomalies between rules in a firewall policy definition. In Sect. 3, we first present our algorithm for detecting and resolving anomalies and its proof of correctness. Then we provide an illustrative example showing how the algorithm works. After that we present our algorithm to merge rules and provide an example of its application. Finally, in Sect. 4, we present the conclusions drawn from our work and propose some directions for future work.

2 Firewall Concepts

In this section, we first discuss the basic concepts of firewall systems and their policy definition. We present our modified definitions of the relationships between the rules in a firewall policy, and then present the anomalies as described in [1].

2.1 Representation of Rules

A rule is defined as a set of criteria and an action to perform when a packet matches the criteria. The criteria of a rule consist of the elements direction, protocol, source IP, source port, destination IP and destination port. Therefore a complete rule may be defined by the ordered tuple ⟨direction, protocol, source IP, source port, destination IP, destination port, action⟩. Each attribute can be defined as a range of values, which can be represented and analyzed as sets.

2.2 Relation between Two Rules

The relation between two rules essentially mean the relation between the set of packets they match. Thus the action field does not come into play when considering the relation between two rules. As the values of the other attributes of firewall rules can be represented as sets, we can consider a rule to be a set of sets, and we can compare two rules using the set relations described in Fig. 1. Two rules can be exactly equal if every criteria in the rules match exactly, one rule can be the subset of the other if each criterion of one rule is a subset of or equal to the other rule's criteria, or they can be overlapped if the rules are not disjoint and at least one of the criteria are overlapped. In the last case, a rule would match a portion of the packets matched by the other but not every

packet, and the other rule would also match a portion of the packets matched by the first rule, but not all.

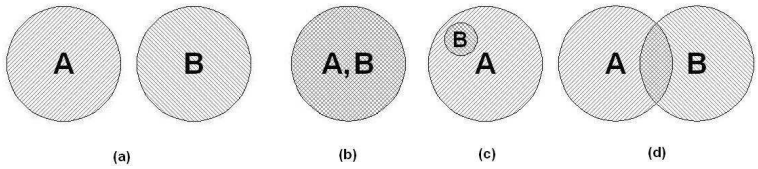

Fig. 1. (a) Sets A and B are disjoint, $A \cap B = \phi$; (b) Sets A and B are equal, $A = B$; (c) Set B is a subset of set A, $B \subset A$; (d) Sets A and B are overlapped, $A \cap B \neq \phi$, but $A \not\subset B$ and $B \not\subset A$.

Al-Shaer et al. discuss these possible relations in [1] and they define the relations *completely disjoint*, *exactly matched*, *inclusively matched*, *partially disjoint* and *correlated*. We propose some modifications to the relations defined in [1]. First we note that it is not needed to distinguish between *completely disjoint* and *partially disjoint* rules as two rules will match entirely different set of packets if they differ in even only in one field. Further, we observe that the formal definition of *correlated* rules does not include the possibility of overlapped field in which the fields are neither disjoint nor subset of one or the other. We propose the following modified set of relations between the rules.

Disjoint Two rules r and s are *disjoint*, denoted as $r \Re_D s$, if they have at least one criterion for which they have completely disjoint values. Formally,
$r \Re_D s$ if $\exists a \in attr[r.a \cap s.a = \phi]$

Exactly Matching Two rules r and s are *exactly matching*, denoted by $r \Re_{EM} s$, if each criterion of the rules match exactly. Formally,
$r \Re_{EM} s$ if $\forall a \in attr[r.a = s.a]$

Inclusively Matching A rule r is a *subset*, or *inclusively matching* of another rule s, denoted by $r \Re_{IM} s$, if there exists at least one criterion for which r's value is a subset of s's value and for the rest of the attributes r's value is equal to s's value. Formally,
$r \Re_{IM} s$ if $\exists_{a \subset attr} [a \neq \phi \wedge \forall_{x \in a} [r.x \subset s.x] \wedge \forall_{y \in a^c} [r.y = s.y]]$

Correlated Two rules r and s are *correlated*, denoted by $r \Re_C s$, if r and s are not disjoint, but neither is the subset of the other. Formally,
$r \Re_C s$ if $(r \not\Re_D s) \wedge (r \not\Re_{IM} s) \wedge (s \not\Re_{IM} r)$

2.3 Possible Anomalies between Two Rules

In [1], Al-Shaer et al. give formal definitions of the possible anomalies between rules in terms of the relations defined in [1]. Of these anomalies, we consider *generalization* not to be an anomaly, as it is used in practice to specially handle a specific group of addresses within a larger group, and as such we omit it from

our consideration. Here, we define the anomalies in terms of the relations in Sect. 2.2.

Shadowing Anomaly A rule r is shadowed by another rule s if s precedes r in the policy, and s can match all the packets matched by r. The effect is that r is never activated. Formally, rule r is shadowed by s if s precedes r, $r\Re_{EM}s$, and $r.action \neq s.action$, or s precedes r, $r\Re_{IM}s$, and $r.action \neq s.action$.

Correlation Anomaly Two rules r and s are correlated if they have different filtering actions and the r matches some packets that match s and the s matches some packets that r matches. Formally rules r and s have a correlation anomaly if $r\Re_C s, r.action \neq s.action$

Redundancy Anomaly A redundant rule r performs the same action on the same packets as another rule s such that if r is removed the security policy will not be affected. Formally rule r is redundant of rule s if s precedes r, $r\Re_{EM}s$, and $r.action = s.action$, or s precedes r, $r\Re_{IM}s$, and $r.action = s.action$; whereas rule s is redundant to rule r if s precedes r, $s\Re_{IM}r$, $r.action = s.action$ and $\nexists t$ where s precedes t and t precedes r, $s\{\Re_{IM}, \Re_C\}t$, $r.action \neq t.action$

3 Anomaly Resolution Algorithms

This section describes the algorithms to detect and resolve the anomalies present in a set of firewall rules as defined in the previous section. The algorithm is in two parts. The first part analyzes the rules and generates a set of disjoint firewall rules that do not contain any anomaly. The second part analyzes the set of rules and tries to merge the rules in order to reduce the number of rules thus generated without introducing any new anomaly.

3.1 Algorithms for Finding and Resolving Anomalies

In this section, we present our algorithm to detect and resolve anomalies. In this algorithm, we resolve the anomalies as follows: in case of *shadowing anomaly*, when rules are *exactly matched*, we keep the one with the reject action. When the rules are *inclusively matched*, we reorder the rules to bring the subset rule before the superset rule. In case of *correlation anomaly*, we break down the rules into disjoint parts and insert them into the list. Of the part that is common to the correlated rules, we keep the one with the reject action. In case of *redundancy anomaly*, we remove the redundant rule.

In our algorithm, we maintain two global lists of firewall rules, *old_rules_list* and *new_rules_list*. The *old_rules_list* will contain the rules as they are in the original firewall configuration, and the *new_rules_list* will contain the output of the algorithm, a set of firewall rules without any anomaly. The approach taken here is incremental, we take each rule in the *old_rules_list* and insert it into *new_rules_list* in such a way that *new_rules_list* remains free from anomalies.

Algorithm RESOLVE-ANOMALIES controls the whole process. After initializing the global lists in lines 1 and 2, it takes each rule from the *old_rules_list* and

invokes algorithm INSERT on it in lines 3 to 4. Then, it scans the *new_rules_list* to resolve any redundancy anomalies that might remain in the list in lines 5 to 10 by looking for and removing any rule that is a subset of a subsequent rule with same action.

Algorithm RESOLVE-ANOMALIES: Resolve anomalies in firewall rules file

```
1.  old_rules_list ← read rules from config file
2.  new_rules_list ← empty list
3.  for all r ∈ old_rules_list do
4.      INSERT(r, new_rules_list)
5.  for all r ∈ new_rules_list do
6.      for all s ∈ new_rules_list after r do
7.          if r ⊂ s then
8.              if r.action = s.action then
9.                  Remove r from new_rules_list
10.             break
```

Algorithm INSERT inserts a rule into the *new_rules_list* in such a way that the list remains anomaly free. If the list is empty, the rule is unconditionally inserted in line 2. Otherwise, INSERT tests the rule with all the rules in *new_rules_list* using the RESOLVE algorithm in the **for** loop in line 5. If the rule conflicts with any rule in the list, RESOLVE will handle it and return *true*, breaking the loop. So, at line 10, if *insert_flag* is *true*, it means that RESOLVE has already handled the rule. Otherwise, the rule is disjoint or superset with all the rules in *new_rules_list* and it is inserted at the end of the list in line 11.

Algorithm INSERT(r,*new_rules_list*): Insert the rule r into *new_rules_list*

```
1.  if new_rules_list is empty then
2.      insert r into new_rules_list
3.  else
4.      inserted ← false
5.      FOR ALL s ∈ new_rules_list  do
6.          if r and s are not disjoint then
7.              inserted ← RESOLVE(r, s)
8.              if inserted = true then
9.                  break
10.     if inserted = false then
11.         Insert r into new_rules_list
```

The algorithm RESOLVE is used to detect and resolve anomalies between two non-disjoint rules. This algorithm is used by the INSERT algorithm. The first rule passed to RESOLVE, r, is the rule being inserted, and the second parameter, s is a rule already in the *new_rules_list*. In comparing them, following are the possibilities:

1. *r and s are equal.* If they are equal, and their actions are same, then any one can be discarded. If the actions are different, then the one with the *reject* action is retained. This case is handled in lines 1 to 6

2. *r is a subset of s.* In this case, we simply insert r before s regardless of the action. This case is handled in lines 7 to 9.
3. *r is a superset of s.* In this case, r may match with rules further down the list, so it is allowed to be checked further. No operation is performed in this case. This case is handled in lines 10 to 11.
4. *r and s are correlated.* In this case, we need to break up the correlated rules into disjoint rules. This case is handled in lines 12 to 19. First the set of attributes in which the two rules differ is determined in line 13, and then SPLIT is invoked for each of the differing attributes in the **for** loop in line 14. After SPLIT returns, r and s contain the common part of the rules, which is then inserted.

Algorithm RESOLVE(r, s): Resolve anomalies between two rules r and s

```
1. if r = s then
2.     if r.action ≠ s.action then
3.         set s.action to REJECT and report anomaly
4.     else
5.         report removal of r
6.     return true
7. if r ⊂ s then
8.     insert r before s into new_rules_list and report reordering
9.     return true
10. if s ⊂ r then
11.    return false
12. Remove s from new_rules_list
13. Find set of attributes a = {x|r.x ≠ s.x}
14. for all a_i ∈ a do
15.    SPLIT(r, s, a_i)
16. if r.action ≠ s.action then
17.    s.action ← REJECT
18. INSERT(s, new_rules_list)
19. return true
```

Algorithm SPLIT, is used to split two non-disjoint rules. It is passed the two rules and the attribute on which the rules differ. It first extracts the parts of the rules that are disjoint to the two rules and invokes the INSERT algorithm on them. Then it computes the common part of the two rules. Let r and s be two rules and let a be the attribute for which SPLIT is invoked. As can be readily seen from the examples in Fig. 2(a) and 2(b), the common part will always start with $\max(r.a.start, s.a.start)$ and end with $\min(r.a.end, s.a.end)$. The disjoint part before the common part begins with $\min(r.a.start, s.a.start)$ and ends with $\max(r.a.start, s.a.start) - 1$, and the disjoint part after the common part starts with $\min(r.a.end, s.a.end) + 1$ and ends with $\max(r.a.end, s.a.end)$. As these two parts are disjoint with r and s, but we do not know their relation with the other rules in *new_rules_list*, they are inserted into the *new_rules_list* by invoking

INSERT procedure. The common part of the two rules is computed in lines 13 and 14. The disjoint part before the common part is computed and inserted in lines 5 to 8. The disjoint part after the common part is computed and inserted in lines 9 to 12.

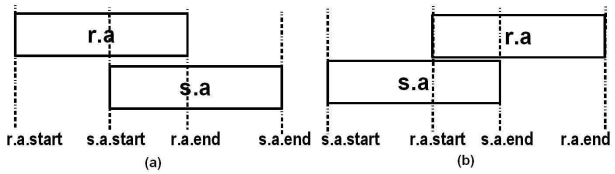

Fig. 2. (a) $r.a.start < s.a.start$ & $r.a.end < s.a.end$, so the ranges can be broken into $[r.a.start, s.a.start - 1]$, $[s.a.start, r.a.end]$ and $[r.a.end + 1, s.a.end]$. (b) $r.a.start > s.a.start$ & $r.a.end > s.a.end$, so the ranges can be broken into $[s.a.start, r.a.start - 1]$, $[r.a.start, s.a.end]$ and $[s.a.end + 1, r.a.end]$.

Algorithm SPLIT(r,s,a): Split overlapping rules r and s based on attribute a

1. $left \leftarrow \min(r.a.start, s.a.start)$
2. $right \leftarrow \max(r.a.end, s.a.end)$
3. $common\_start \leftarrow \max(r.a.start, s.a.start)$
4. $common\_end \leftarrow \min(r.a.end, s.a.end)$
5. **if** $r.a.start > s.a.start$ **then**
6. INSERT($\langle(left, common\_start - 1)$, rest of s's attributes$\rangle$, new_rules_list)
7. **else if** $r.a.start < s.a.start$ **then**
8. INSERT($\langle(left, common\_start - 1)$, rest of r's attributes$\rangle$, new_rules_list)
9. **if** $r.a.end > s.a.end$ **then**
10. INSERT($\langle(common\_end + 1, right)$, rest of r's attributes$\rangle$, new_rules_list)
11. **else if** $r.a.end < s.a.end$ **then**
12. INSERT($\langle(common\_end + 1, right)$, rest of s's attributes$\rangle$, new_rules_list)
13. $r \leftarrow \langle(common\_start, common\_end)$, rest of r's attributes$\rangle$
14. $s \leftarrow \langle(common\_start, common\_end)$, rest of s's attributes$\rangle$

After completion of the RESOLVE-ANOMALIES algorithm, *new_rules_list* will contain the list of firewall rules that are free from all the anomalies in consideration.

3.2 Proof of Correctness

To prove the correctness of our algorithm, we first present and prove the following theorem.

Theorem 1. *A set of rules* $R[1 \ldots n]$ *is free from* shadowing, correlation *and* redundancy *anomalies if for* $1 \leq i < j \leq n$, *exactly one of the following three conditions hold:*

1. $R[i]\Re_D R[j]$
2. $R[i]\Re_{IM} R[j]$ *and* $R[i].action \neq R[j].action$
3. $R[i]\Re_{IM} R[j]$ *and* $R[i].action = R[j].action$ *only if there exists some* k *such that* $i < k < j$ *and* $R[i]\Re_{IM} R[k]$ *and* $R[i].action \neq R[k].action$

Proof. *Shadowing anomaly* cannot exist in the list as for $R[i]$ cannot be a subset of $R[j]$ if $i > j$. *Correlation anomaly* cannot exist in the list as the only relations allowed in the list are $\Re_D$ and $\Re_{IM}$. The absence of *redundancy anomaly* is ensured by conditions 2 and 3. □

We show by using loop invariants [6] that after the completion of Algorithm Resolve-Anomalies, *new_rules_list* will maintain these properties and so it will be anomaly free. The loop invariant is:

> At the start of each iteration of the **for** loop in line 3 of Algorithm Resolve-Anomalies, for $1 \leq i < j \leq m$, exactly one of the following holds:
> 1. $new\_rules\_list[i]\Re_D new\_rules\_list[j]$
> 2. $new\_rules\_list[i]\Re_{IM} new\_rules\_list[j]$
>
> where m is the size of *new_rules_list*.

Initialization Before the first iteration, the *new_rules_list* is initialized to *empty* and the invariant holds trivially.

Maintenance Let r be the rule being inserted in an iteration. When Insert is invoked on r, the following cases are possible:

1. *new_rules_list* is empty. Then r is inserted into *new_rules_list*, and the iteration is complete, with the invariant holding trivially.
2. r is disjoint with every rule in *new_rules_list*. In this case, r is inserted at the end of the list. The iteration is complete, and the loop invariant holds.
3. r is not disjoint with rule $s \in new\_rules\_list$. Then the procedure Resolve is invoked with r and s, which checks the following possibilities:
   (a) $r\Re_{EM} s$. In this case, if their actions are same, r is redundant and need not be inserted. If their actions are different, then r and s are conflicting rules, and we simply change the action of s to *reject*, without inserting r. In both cases, the iteration completes without inserting anything, so the loop invariant holds.
   (b) $r\Re_{IM} s$. In this case, r is inserted before s in *new_rules_list* . The iteration is complete, and the loop invariant holds.
   (c) $s\Re_{IM} r$. In this case, Resolve returns *false* so that the loop in line 5 in Insert can continue until either the end of *new_rules_list* has been reached, or for some subsequent rule t has been found that is not disjoint with r and call to Resolve(r, t) has returned *true*. In the first case, all the rules after s in *new_rules_list* are disjoint with r, so r can be appended to *new_rules_list* without violating the loop invariant. In the second case, the call to Resolve(r, t) has handled r without violating the loop invariant, so in both cases the invariant holds.

(d) $r \Re_C s$. In this case, first s is removed from *new_rules_list* as r and s are going to be broken into disjoint rules. Then, algorithm SPLIT is invoked for each attribute for which r and s differ. SPLIT breaks up the attribute's value into the part common to the rules and the parts unique to the rules. The unique parts are by definition disjoint, so they are inserted into the list by calling INSERT. After breaking up all the non-matching attributes, r and s are exactly matched. If they are of different action, then *s.action* is set to *reject*, otherwise s already contains the common action. Then s is inserted into *new_rules_list* by INSERT procedure. This completes the iteration, and ensures that the loop invariant holds.

Termination The loop terminates when all the rules in *old_rules_list* has been inserted into *new_rules_list*, and as the loop invariant holds for each iteration, we have that conditions of the loop invariant hold for the entire *new_rules_list*.

Thus, at the end of the **for** loop in line 3 of Algorithm RESOLVE-ANOMALIES, each element of the *new_rules_list* is either disjoint with or subset of the subsequent elements. The **for** loop at line 5 scans and removes any redundancy anomalies present in the *new_rules_list*. If any rule r in *new_rules_list* is subset of any subsequent rule s in *new_rules_list* with the same action, r is removed as it is redundant. So at the end of this loop, *new_rules_list* will maintain the three properties stated in Theorem 1, and so *new_rules_list* will be free from *shadowing*, *correlation* and *redundancy* anomalies.

3.3 Cost Analysis

The cost of running the algorithm basically depends on the nature of the rules in the input. If a rule is disjoint or superset with the other rules of the *new_rules_list* then the rule is inserted into the list without further invoking procedure INSERT. In this case the **for** loop in line 5 of Algorithm INSERT has to traverse the whole *new_rules_list*. If a rule is subset of any rule in the *new_rules_list*, then the rule is inserted just before the superset rule and the loop terminates immediately. So in the worst case the whole *new_rules_list* may have to be traversed. A rule is discarded if it is equal to any rule in the *new_rules_list*, and in the worst case the whole *new_rules_list* may have to be traversed. If a rule is correlated with a rule in the *new_rules_list* then they will be divided into a set of mutually disjoint rules. In the worst case the number of rules thus generated will be up to twice the number of attributes plus one, and these rules will be inserted into the *new_rules_list* by invoking INSERT recursively. RESOLVE-ANOMALIES invokes INSERT once for each rule in *new_rules_list* in the **for** loop in line 3, and removes the redundancy anomalies in the **for** loop in line 5. The running time of RESOLVE-ANOMALIES is dominated by the number of times INSERT is invoked, which depends on the number of correlated rules. In the worst case, if all rules are correlated, the running time may deteriorate to exponential order.

3.4 Illustrative Example

Let us consider the following set of firewall rules for analysis with the algorithm.

1. ⟨IN, TCP, 129.110.96.117, ANY, ANY, 80, REJECT⟩
2. ⟨IN, TCP, 129.110.96.*, ANY, ANY, 80, ACCEPT⟩
3. ⟨IN, TCP, ANY, ANY, 129.110.96.80, 80, ACCEPT⟩
4. ⟨IN, TCP, 129.110.96.*, ANY, 129.110.96.80, 80, REJECT⟩
5. ⟨OUT, TCP, 129.110.96.80, 22, ANY, ANY, REJECT⟩
6. ⟨IN, TCP, 129.110.96.117, ANY, 129.110.96.80, 22, REJECT⟩
7. ⟨IN, UDP, 129.110.96.117, ANY, 129.110.96.*, 22, REJECT⟩
8. ⟨IN, UDP, 129.110.96.117, ANY, 129.110.96.80, 22, REJECT⟩
9. ⟨IN, UDP, 129.110.96.117, ANY, 129.110.96.117, 22, ACCEPT⟩
10. ⟨IN, UDP, 129.110.96.117, ANY, 129.110.96.117, 22, REJECT⟩
11. ⟨OUT, UDP, ANY, ANY, ANY, ANY, REJECT⟩

Step-1. As the *new_rules_list* is empty, rule-1 is inserted as it is.

Step-2. When rule-2 is inserted, *new_rules_list* contains only one rule, the one that was inserted in the previous step. We have, r = ⟨IN, TCP, 129.110.96.*, ANY, ANY, 80, ACCEPT⟩ and s = ⟨IN, TCP, 129.110.96.117, ANY, ANY, 80, REJECT⟩. Here, $s \subset r$, so r is inserted into *new_rules_list* after s.

Step-3. In this step, r = ⟨IN, TCP, ANY, ANY, 129.110.96.80, 80, ACCEPT⟩. In the first iteration, s = ⟨IN,TCP,129.110.96.117,ANY,ANY,80,REJECT⟩. Clearly these two rules are correlated, with $s.srcip \subset r.srcip$ and $r.destip \subset s.destip$. Therefore these rules must be broken down. After splitting the rules into disjoint parts, we have the following rules in *new_rules_list*:

   1. ⟨IN, TCP, 129.110.96.1-116, ANY, 129.110.96.80, 80, ACCEPT⟩
   2. ⟨IN, TCP, 129.110.96.118-254, ANY, 129.110.96.80, 80, ACCEPT⟩
   3. ⟨IN, TCP, 129.110.96.117, ANY, 129.110.96.1-79, 80, REJECT⟩
   4. ⟨IN, TCP, 129.110.96.117, ANY, 129.110.96.81-254, 80, REJECT⟩
   5. ⟨IN, TCP, 129.110.96.117, ANY, 129.110.96.80, 80, REJECT⟩
   6. ⟨IN, TCP, 129.110.96.*, ANY, ANY, 80, ACCEPT⟩

After completion of the first **for** loop in line 3 in Algorithm Resolve-Anomalies, the *new_rules_list* will hold the following rules:

1. ⟨IN, TCP, 129.110.96.1-116, ANY, 129.110.96.80, 80, ACCEPT⟩
2. ⟨IN, TCP, 129.110.96.118-254, ANY, 129.110.96.80, 80, ACCEPT⟩
3. ⟨IN, TCP, 129.110.96.117, ANY, 129.110.96.1-79, 80, REJECT⟩
4. ⟨IN, TCP, 129.110.96.117, ANY, 129.110.96.81-254, 80, REJECT⟩
5. ⟨IN, TCP, 129.110.96.117, ANY, 129.110.96.80, 80, REJECT⟩
6. ⟨IN, TCP, 129.110.96.*, ANY, 129.110.96.80, 80, REJECT⟩
7. ⟨IN, TCP, 129.110.96.*, ANY, ANY, 80, ACCEPT⟩
8. ⟨OUT, TCP, 129.110.96.80, 22, ANY, ANY, REJECT⟩
9. ⟨IN, TCP, 129.110.96.117, ANY, 129.110.96.80, 22, REJECT⟩
10. ⟨IN, UDP, 129.110.96.117, ANY, 129.110.96.80, 22, REJECT⟩
11. ⟨IN, UDP, 129.110.96.117, ANY, 129.110.96.117, 22, REJECT⟩

12. ⟨IN, UDP, 129.110.96.117, ANY, 129.110.96.*, 22, REJECT⟩
13. ⟨OUT, UDP, ANY, ANY, ANY, ANY, REJECT⟩

The next step is to scan this list to find and resolve the redundancy anomalies. In this list, rule-1 is a subset of rule-6, but as the rules have different action, rule-1 is retained. Similarly, rule-2, which is also a subset of rule-6 with differing action, is also retained. Rules 3 and 4 are subsets of rule-7, but are retained as they have different action than rule-7. Rule-5 is a subset of rule-6, and as they have the same action, rule-5 is removed. After removing these rules, the list is free from all the anomalies.

3.5 Algorithms for Merging Rules

After the completion of the anomaly resolution algorithm, there are no correlated rules in the list. In this list, we can merge rules having attributes with consecutive ranges with the same action. To accomplish this, we construct a tree using Algorithm TreeInsert. Each node of the tree represents an attribute. The edges leading out of the nodes represent values of the attribute. Each edge in the tree represents a particular range of value for the attribute of the source node, and it points to a node for the next attribute in the rule represented by the path. For example, the root node of the tree represents the attribute *Direction*, and there can be two edges out of the root representing *IN* and *OUT*. We consider a firewall rule to be represented by the ordered tuple as mentioned in Sect. 2.1. So, the edge representing the value *IN* coming out of the root node would point to a node for *Protocol*. The leaf nodes always represent the attribute *Action*. A complete path from the root to a leaf corresponds to one firewall rule in the policy. For example, the leftmost path in the tree in Fig. 3(a) represents the firewall rule ⟨IN, TCP, 202.80.169.29-63, 483, 129.110.96.64-127, 100-110, ACCEPT⟩.

Algorithm TreeInsert, takes as input a rule and a node of the tree. It checks if the value of the rule for the attribute represented by the node matches any of the values of the edges out of the node. If it matches any edge of the node, then it recursively invokes TreeInsert on the node pointed by the edge with the rule. Otherwise it creates a new edge and adds it to the list of edges of the node.

Algorithm TreeInsert(n, r): Inserts rule r into the node n of the rule tree

1. **for all** *edge* $e_i \in n.edges$ **do**
2. **if** $r.(n.attribute) = e_i.range$ **then**
3. TreeInsert($e_i.vertex$, r)
4. **return**
5. $v \leftarrow$ new Vertex(next attribute after $n.attribute$, NULL)
6. Insert new edge $\langle r.(n.attribute), r.(n.attribute), \text{v} \rangle$ in $n.edges$
7. TreeInsert(v, r)

We use Algorithm MERGE on the tree to merge those edges of the tree that has consecutive values of attributes, and has exactly matching subtrees. It first calls itself recursively on each of its children in line 2 to ensure that their subtrees are already merged. Then, it takes each edge and matches its range with all the other edges to see if they can be merged. Whether two edges can be merged depends on two criteria. First, their ranges must be contiguous i.e. the range of one starts immediately after the end of the other. Second, the subtrees of the nodes pointed to by the edges must match exactly. This criterion ensures that all the attributes after this attribute are same for all the rules below this node. If these two criteria are met, they are merged into one edge in place of the original two edges. After merging the possible rules, the number of rules defined in the firewall policy is reduced and it helps to increase the efficiency of firewall policy management. The example given in Fig. 3 illustrate how the merge procedure works.

Algorithm MERGE(n): Merges edges of node n representing a continuous range

1. **for all** *edge* $e \in n.edges$ **do**
2. MERGE($e.node$)
3. **for all** *edge* $e \in n.edges$ **do**
4. **for all** *edge* $e' \neq e \in n.edges$ **do**
5. **if** e's and e''s ranges are *contiguous* and Subtree(e)=Subtree(e') **then**
6. Merge $e.range$ and $e'.range$ into $e.range$
7. Remove e' from $n.edges$

3.6 Illustrative Example of the Merge Algorithm

To illustrate the merging algorithm, we start with the following set of non-anomalous rules. We deliberately chose a set of rules with the same action since rules with different action will never be merged.

1. ⟨IN, TCP, 202.80.169.29-63, 483, 129.110.96.64-127, 100-110, ACCEPT⟩
2. ⟨IN, TCP, 202.80.169.29-63, 483, 129.110.96.64-127, 111-127, ACCEPT⟩
3. ⟨IN, TCP, 202.80.169.29-63, 483, 129.110.96.128-164, 100-127, ACCEPT⟩
4. ⟨IN, TCP, 202.80.169.29-63, 484, 129.110.96.64-99, 100-127, ACCEPT⟩
5. ⟨IN, TCP, 202.80.169.29-63, 484, 129.110.96.100-164, 100-127, ACCEPT⟩
6. ⟨IN, TCP, 202.80.169.64-110, 483-484, 129.110.96.64-164, 100-127,ACCEPT⟩

From this rules list we generate the tree as in Fig 3(a) by the TREEINSERT algorithm. On this tree, the Merge procedure is run. The Merge algorithm traverses the tree in post order. Thus, the first node to be processed is node 14. As it has only one child, it returns without any operation. The next node in order is 15, which also has only one child. The next node to be processed is node 9. The attribute represented by node 9 is destination port. The ranges of its two children, 100-110 and 111-127 are consecutive, and also their subtrees are the same. Thus, these two edges are merged to obtain one edge with the range 100-127.

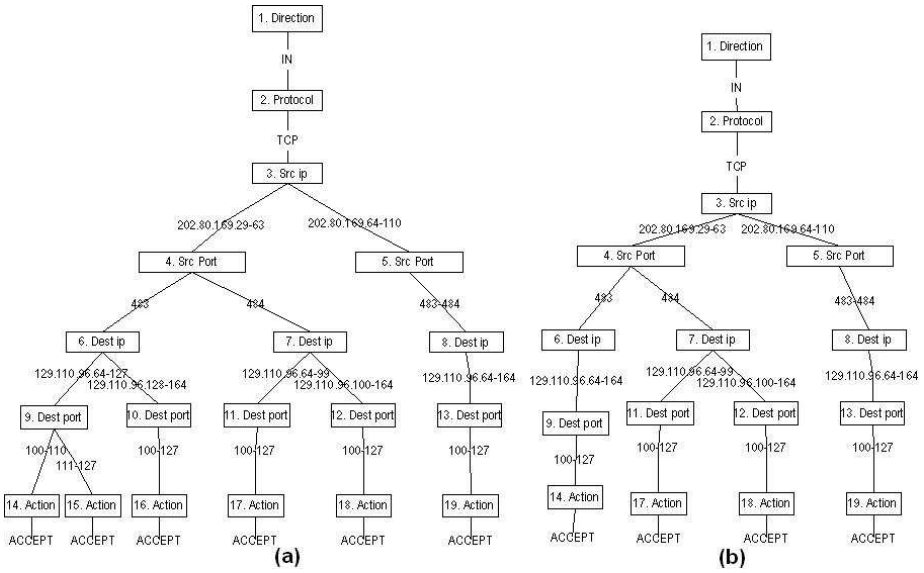

Fig. 3. (a) Tree generated from the example rules list by the TreeInsert algorithm. (b) Intermediate state of the tree: node 7's children are going to be merged next.

Of the nodes that are going to be processed now, nodes 16 and 10 are each of one child only, so they do not require any further processing. However, node 6, representing destination IP address, has two children, and their IP address ranges, 129.110.96.64-127 and 129.110.96.128-164, are consecutive. Also, the subtrees from the nodes 9 and 10 are also the same. Hence, they are merged to become one edge with range value of 129.110.96.128-164.

So far, we have eliminated three nodes, 15, 16 and 10. Of the next nodes to be processed, nodes 17, 11, 18 and 12 has only one child, and hence incur no processing. At node 7, as shown in Fig 3(b), the represented attribute is destination IP address, and the ranges of its children are 129.110.96.64-99 and 129.110.96.100-164. The ranges being consecutive, they are merged into one edge of range 129.110.96.64-164, eliminating node 12 and 18.

Continuing in this way, we are left with the single rule ⟨IN,TCP,202.80.169.29-110, 483-484,129.110.96.64-164,100-127,ACCEPT⟩ after the merging is complete on the entire tree.

4 Conclusion and Future Works

Resolution of anomalies from firewall policy rules is vital to the network's security as anomalies can introduce unwarranted and hard to find security holes. Our work presents an automated process for detecting and resolving such anomalies. The anomaly resolution algorithm and the merging algorithm should produce a compact yet anomaly free rule set that would be easier to understand and

maintain. This algorithms can also be integrated into policy advisor and editing tools. The paper also establishes the complete definition and analysis of the relations between rules.

In future, this analysis can be extended to distributed firewalls. Also, we propose to use data mining techniques to analyze the log files of the firewall and discover other kinds of anomalies. These techniques should be applied only after the rules have been made free from anomaly by applying the algorithms in this paper. That way it would be ensured that not only syntactic but also semantic mistakes in the rules will be captured. Research in this direction has already started.

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] E. Al-Shaer and H. Hamed. Firewall policy advisor for anomaly detection and rule editing. In *IEEE/IFIP Integrated Management Conference (IM'2003)*, March 2003.

[3] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proc. 23rd Conf. IEEE Communications Soc. (INFOCOM 2004), Vol. 23, No. 1*, pages 2605–2616, March 2004.

[4] E. Al-Shaer and H. Hamed. Taxonomy of conflicts in network security policies. *IEEE Communications Magazine*, 44(3), March 2006.

[5] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan. Conflict classification and analysis of distributed firewall policies. *IEEE Journal on Selected Areas in Communications (JSAC)*, 23(10), October 2005.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, U.S.A, 2nd edition, 2001.

[7] D. Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *Proceedings of the 12th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA 2001)*, pages 827–835, January 2001.

[8] P. Eronen and J. Zitting. An expert system for analyzing firewall rules. In *Proceedings of the 6th Nordic Workshop on Secure IT Systems (NordSec 2001)*, pages 100–107, November 2001.

[9] Z. Fu, S. 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 Policy2001 Workshop*, January 2001.

[10] K. Golnabi, R. K. Min, L. Khan, and E. Al-Shaer. Analysis of firewall policy rules using data mining techniques. In *IEEE/IFIP Network Operations and Management Symposium (NOMS 2006)*, April 2006.

[11] A. Hari, S. Suri, and G. M. Parulkar. Detecting and resolving packet filter conflicts. In *INFOCOM (3)*, pages 1203–1212, March 2000.

[12] S. Hazelhurst. Algorithms for analysing firewall and router access lists. Technical Report TR-WitsCS-1999-5, Department of Computer Science, University of the Witwatersrand, South Africa, July 1999.

[13] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *Proceedings, IEEE Symposium on Security and Privacy*, pages 177–187. IEEE CS Press, May 2000.