Fault Detection in Rule-Based Software Systems

Dong Wang, Ruibing Hao and David Lee

Bell Labs Research China
Beijing, China, 100080
{wangd, rbao, leedavid}@lucent.com

Abstract

Motivated by packet filtering of firewall systems in Internet applications, we study the fault detection problem in the general rule-based software systems. We discuss algorithms for the detection of conflicts in a given set of rules. We first study a constrained version of the fault detection problem and propose a two-phase algorithm. The first phase is to do the rule normalization. The second phase is to detect conflicting rules. For this constrained version of the fault detection problem, the algorithm takes polynomial time. For the general problem, it is NP-hard. We apply the algorithms to the rule table getting from one of the firewalls in Bell Labs and report the experiment result.

Keywords

Rule-Based Software system, fault detection, Packet Filter

1 Introduction

With the complexity and dynamic nature of communication systems, fault detection is the center of focus in communication network management and yet has proved to be a formidable task. Fault detection problem has been investigated heavily by many researchers from various aspects in the past. Existing fault detection approaches can be classified into the following categories according to the techniques they use: expert systems, finite state machines, advanced database techniques, and probabilistic approaches [1]. The following papers offer a glimpse of the arena [2][3][4]. Fault detection is also a more complex problem when compared with the general conformance testing, in that the latter just needs to check if an implementation is different from its specification, while the former needs to locate the difference found [5]. In this paper, we investigate the fault detection problem from a different perspective, or say the fault detection problem in a rule-based communication software system, which has not been formally investigated before.

In modern communication software systems, Rule-Based Software systems (RBS) are commonly used where program actions are determined by a group of pre-defined rules. Its desired functioning largely depends on the correctness and coherence of all the rules. However, because the rules in an RBS are configured manually, conflicts among these rules may lead to unexpected program actions from the RBS.

For example, a typical scenario of conflict is that for some specific inputs of the RBS, multiple rules are eligible and can be applied. While the rule selected by the RBS is not what the designer or administrator has expected. So it is necessary to detect and remove conflicts among rules of an RBS to prevent the above scenario from happening. We call this problem the fault detection problem in RBS. When the rule set is small, for example four to five rules, it is easy to detect all the conflict manually. But it is extremely hard to find out the conflicts manually when the number of rules in the RBS grows very large. So we need to investigate how to do fault detection for an RBS
automatically.

This paper presents a method to solve this problem for the packet filtering software in the firewall system, which is a typical type of RBS. We first investigate a constrained version of this fault detection problem and then relax the constraints to find more general solutions. We propose a two-phase algorithm to solve this constrained version of fault detection problem. The first phase is to do the rule normalization. The second phase is to detect conflicting rules. For this constrained version of the fault detection problem, the algorithm takes polynomial time.

We also discuss solutions for the more general fault detection problem, in which the above two assumptions do not hold. For RBS that has multiple actions in the action part of each rule, there is a polynomial algorithm to check if the actions of two rules conflict. If the assumption about the conditional part does not hold, we need to solve a more general problem: to decide whether or not the intersection of two conditional expressions is satisfiable. We believe it is an NP-complete problem by reducing the well-known SAT problem to this problem.

This paper is an extended version of [10], it is organized as follows: The fault detection problem of RBS is defined in Section 2 by using the packet filters in firewall systems as an example. In section 3, we present our solution to a constrained version of the fault detection problem. In section 4 we extend our solution to deal with composite-actions. In section 5 we present that in general the fault detection problem is an NP-complete problem. In section 6, we do a case study on the data collected from a real firewall system in Bell Labs and report the experiment result. Section 7 concludes this paper with future works.

2 Fault Detection Problem for Packet Filters – a Case Study

Network security is one of the most important concerns of enterprise networks connected to the Internet. To prevent malicious denial-of-service attack and unauthorized access to or from a private network, firewall systems have been widely deployed in enterprise networks to enhance network security by separating the private Intranet from the public Internet. All messages entering or leaving the intranet pass through the firewall, which examines each message and blocks those that do not meet the specified security criteria.

There are several types of firewall techniques: packet filter, application gateway, circuit-level gateway, and proxy server [6]. Packet filter is one of the most effective firewall techniques. A packet filter looks at each packet entering or leaving the network and accepts or rejects it based on user-defined rules. Packet filtering is fairly effective and transparent to users, but it is difficult to configure.

2.1 Rules in a Packet Filter

Packet filter is a typical type of RBS. Traditionally a packet filter examines packets at IP layer, or TCP/UDP layer, or both. Usually it checks the values of the five fields in the TCP/IP header, including destination address, source address, destination port, source port and transport layer protocol type, against a set of filtering rules. A rule in the packet filter usually consists of two parts: the condition part and the action parts. It describes the actions the packet filter will take when the packet satisfies or fails the logical expression in the condition part. For example,

\[ \text{If (packet's destination port equals 21) then (discard it) else (forward it)} \]  \hspace{1cm} (1)
Rule (1) specifies a pass-action “discard it” and a fail-action “forward it” for conditional expression “destination port equals 21”. When the Boolean value of the conditional expression is true, the packet filter will take the pass-action; otherwise the fail-action. The fail-action part is optional.

A packet filter usually has more than one rules, all the rules in a packet filter form a so-called Rule Table (RT). Rules in a rule table can be chained together by a special action goto. Table 1 shows an example rule table. For each received packet, the packet filter works as follows. It will test the packet against the rules according to the order that rules are listed in the rule table. The first rule in the rule table will be checked first. If the packet satisfies the conditional expression of the rule, the pass-action of this rule will be taken; otherwise the fail-action will be taken. If the packet fails the current conditional expression and the fail-action part is not presented in the rule, the next rule in the rule table will be checked. If the action taken by the packet filter is a goto action, then the next rule pointed by goto will be checked. If the checking falls off the table then the packet is discarded.

<table>
<thead>
<tr>
<th>No.</th>
<th>Content of rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>If (packet’s destination port equals 21) then (discard it)</td>
</tr>
<tr>
<td>R2</td>
<td>If (packet’s destination port small than 255) then (goto3) else (discard it)</td>
</tr>
<tr>
<td>R3</td>
<td>If (packet’s source is subnet 166.111.0.0/16) then (forward it)</td>
</tr>
</tbody>
</table>

Table 1. Example Rule Table

2.2 Fault Detection Problem

The desired functioning of a packet filter largely depends on the correctness and coherence of all the rules. But because the rules in a packet filter are configured manually, conflicts among these rules may lead to unexpected actions from the packet filter.

For example, in Table 1 the goal of R1 is to discard all the packets with destination port 21. R2 is not a complete rule for its pass-action is “goto 3”. This means that R2 must be used in combination with R3. R3 is to forward all the packets from subnet 166.111.0.0/16, so the combined result of R2 and R3 is to forward all the packets coming from subnet 166.111.0.0/16 and with destination port small than 255.

There is a hidden conflict in this rule table. Assume a packet arrives with destination port 21, which is obviously small than 255, and source address 166.111.68.1. Certainly if the packet filter starts the checking process from rule R1, the packet will be discarded. But this may not be the result the rule designer or the administrator has expected.

There are many reasons why this kind of hidden conflicts exists. The most reasonable one is that the rule designer does not have a clear and complete understanding of the former configured rules when adding new rules to the rule table. This situation actually happens quite often when the number of rules in the rule table is very large. So it is necessary to detect these hidden conflicts among rules. If conflicts among rules exist, rule designer can refine the conflicting rules in case an undesired action is taken.

We call the problem of detecting conflicts among rules in an RBS as the fault detection problem of RBS. When the rule set is small, for example four to five rules, it is easy to detect all the conflicts manually. But it is extremely hard to do it manually when the number of rules in the RBS grows very large. So we need to investigate how to do fault detection for an RBS automatically.
3 Solution to a Constrained Version of Fault Detection Problem

In this section, we first give the definition of rules and conflict rules, and then investigate the solution to a constrained version of the fault detection problem. We propose a two-phase algorithm to solve this constrained version of fault detection problem. The first phase is to do the rule normalization. The second phase is to detect conflict rules. For this constrained version of the fault detection problem, the algorithm takes polynomial time.

3.1 Conflict Rules

Suppose all rules in the rule table of a packet filter can be converted into the following format,

\[
\text{if } (C) \text{ then } (A),
\]

(2)

\(C\) is a conditional expression and \(A\) is pass-action part of the rule. We will show how to do the rule conversion in next section. Conflict rules can be defined as:

Definition 1. Conflict rules

Given a rule table of a packet filter, which has \(N\) pieces of rules,

\[R_i : \text{if } (C_i) \text{ then } (A_i) \quad 1 \leq i \leq N\]

We say that two pieces of rules \(R_i\) and \(R_j\) conflict if and only if \(A_i\) conflicts with \(A_j\) and \(C_i \land C_j\) is satisfiable. □

So the fault detection problem in an RBS such as packet filter is to find all possible conflicts in its rule table.

3.2 Rule normalization

As we can see from Table 1, there exist many rules that do not match the general if-then format as in (2). Usually these rules can be expressed as following,

\[
\text{If } (C_i) \text{ then } (A_{i\text{if}}) \text{ else } (A_{i\text{else}})
\]

(3)

So we need to do rule normalization to convert rules with the if-then-else format as in (3) to the required if-then format as in (2).

As we pointed out in Section 2, rule designers often use a special action “goto” to chain rules together, which makes the configuration of packet filter easier and more flexible. But for a rule with the special action “goto”, we cannot decide its final action directly. We need to further check the rule linked by the “goto” action. This will lead to trouble when doing fault detection. So another task of rule normalization is to get rid of all the “goto” action.

The basic idea of rule normalization is to represent the rule table by a directed graph, which is called as the Rule Relation Graph (RRG). Then every path from the source node to the sink node in the RRG forms a new rule, which is free of “goto” and “else”.
**Definition 2. Rule Relation Graph**

A Rule Relation Graph (RRG) is a graph $G = (V, E)$ in which,

- $V = \{C_1, C_2, ..., C_N, A_1, A_2, ..., A_N, S\}$ is the node set of graph $G$, in which $S$ is a pseudo source node; $C_i$ represents a condition node; $A_i$ represents an action node. Every action node $A_i$ is also a sink node.
- $E$ is the edge set of graph $G$. Every edge in $E$, except the edges starting from the pseudo source node $S$, has a label, either YES or NO.

The algorithm in Figure 1 shows how to form the RRG from a rule table, whose time complexity is $O(N)$, the value of $N$ ranges from tens to hundreds in practice.

```
Algorithm 1: Forming RRG
Input: Rule table with $N$ rules
Output: RRG $G$
1 $V = \{S\}; E = \varnothing$; set each rule as "unvisited";
2 For each $R_i = (C_i, A_{yi}, A_{ni}), i = 1$ to $N$
3 if rule $R_i$ is "unvisited"
4 set rule $R_i$ as "visited";
5 $V = V \cup \{\text{node } C_i\}$;
6 $E = E \cup \{\text{edge from } S \text{ to node } C_i\}$;
7 if ($A_{yi}$ is "goto $j$")
8 if rule $R_j$ is unvisited
9 $V = V \cup \{\text{node } C_j\}$;
10 $E = E \cup \{\text{edge from node } C_i \text{ to node } C_j \}$ with label YES;
11 else
12 $V = V \cup \{\text{node } A_{ni}\}$;
13 $E = E \cup \{\text{edge from node } C_i \text{ to node } A_{ni} \}$ with label YES;
14 if ($A_{ni}$ is "goto $k$")
15 if rule $R_k$ is unvisited
16 $V = V \cup \{\text{node } C_k\}$;
17 $E = E \cup \{\text{edge from node } C_i \text{ to node } C_k \}$ with label NO;
18 else
19 $V = V \cup \{\text{node } A_{yi}\}$;
20 $E = E \cup \{\text{edge from node } C_i \text{ to node } A_{yi} \}$ with label NO;
```

Figure 1. Algorithm for forming RRG

For example, the RRG in Figure 2 corresponds to rule table in Table 1.
Then every source to sink path in RRG represents a new rule. Every path can be rewritten into a new rule satisfying the standard if-then format by following the procedure below:

- The sink node of the path forms the action part of the new rule;
- The conjunction of all the intermediate nodes forms the condition part of the new rule, the conjunction is generated as follows: if the edge coming out of node \( M \) is labeled with “YES”, we use node \( M \) itself in the condition part, otherwise we use negation of \( M (\overline{M}) \) instead.

From Figure 2, three paths can be found.

1) \( S \rightarrow (DP = 21) \xrightarrow{YES} \text{(Discard)} \)
2) \( S \rightarrow (DP < 255) \xrightarrow{NO} \text{(Discard)} \)
3) \( S \rightarrow (DP < 255) \xrightarrow{YES} (SD = 166^*) \xrightarrow{YES} \text{(Forward)} \)

So three new rules can be generated from Figure 2, as shown in Table 2.

<table>
<thead>
<tr>
<th>No.</th>
<th>Content of rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>If (packet with destination port equals 21) then (discard it)</td>
</tr>
<tr>
<td>2</td>
<td>If (packet with destination port large than or equals 255) then (discard it)</td>
</tr>
<tr>
<td>3</td>
<td>If (packet with destination port small than 255 and from subnet 166.111.0.0/16) then (forward it)</td>
</tr>
</tbody>
</table>

Table 2. Normalized Rule Table

### 3.3 Fault Detection

The second phase of our solution is to detect conflicting rules in the normalized rule table. We first check if the action part of every pair of rules conflict. If they do, we then check if the intersection of the two conditional expressions is satisfiable. If yes, a conflict is found. To make this problem easier to solve, we first make two assumptions. Later we will relax these two assumptions and discuss general solutions.

The first assumption is that there is only one action in every piece of rule, for example, in packet filter, either forward or discard, and these two conflict. So in the rule table, different actions means conflict.
The second assumption is about the conditional expressions of rules. We assume that we have limited number (K) of variables in the conditional expressions, $X_i, 1 \leq i \leq K$. Each variable corresponds to a dimension in a K-dimension space. And the possible values in each dimension are limited in a finite, continuous and ordered integer interval represented by $S_{Xi}$.

We call the general fault detection problem with these two assumptions as a constrained version of the fault detection problem.

For packet filter, five fields in the TCP/IP header (see section 2.1) have been used in the conditional expressions. Suppose we use five dimensions, $da, sa, dp, sp$ and $tp$, to represent these five fields. The values of each field form an interval met the above condition in the corresponding dimension. And an arbitrary sub-interval of each of the five dimensions can be denoted by $S_{da}, S_{sa}, S_{dp}, S_{sp}$ and $S_{tp}$ respectively.

Given an arbitrary conditional expression $C$, we can transform it into a disjunctive normal form [7]:

$$F = (S_{da}^1 \land S_{sa}^1 \land S_{dp}^1 \land S_{sp}^1 \land S_{tp}^1) \lor (S_{da}^2 \land S_{sa}^2 \land S_{dp}^2 \land S_{sp}^2 \land S_{tp}^2) \lor \ldots \lor (S_{da}^{n} \land S_{sa}^{n} \land S_{dp}^{n} \land S_{sp}^{n} \land S_{tp}^{n})$$

(4)

Because $F$ is a conditional expression in disjunctive normal form, we can make sure every interval $S_{x}^i, x \in \{da, sa, dp, sp, tp\}$ in expression 4 is a single interval in the corresponding dimension. Single interval means that for every possible value $e$ in dimension $x$, if $\text{MIN}(S_{x}^i) < e < \text{MAX}(S_{x}^i)$, then $e \in S_{x}^i$.

Figure 3 contains an algorithm to check the satisfiability of the intersection of two conditional expressions $C_1$ and $C_2$ in disjunctive normal form. For two intervals $S_{x}^{m}$ and $S_{y}^{k}$ in different dimensions, there is no need to check for intersection. So we only need to consider intervals belonging to the same dimension. In the algorithm, we call every conjunctive form a clause, e.g. $(S_{da}^{i} \land S_{sa}^{i} \land S_{dp}^{i} \land S_{sp}^{i} \land S_{tp}^{i})$ is a clause.
Algorithm 2: Checking satisfiability

| Input: | Two condition expressions $C_1$ and $C_2$ in disjunctive normal form |
| Output: | Boolean value to show if there is intersection between $C_1$ and $C_2$ |

1. For each clause in $C_1$
2. For each clause in $C_2$
3. For each dimension in set {da, sa, dp, sp, tp}
4. If both clauses have single intervals in current dimension
5. Check if these two single intervals have intersection
6. If yes, return TRUE;
7. Return FALSE

Figure 3. Algorithm for checking satisfiability

It is easy to prove that the algorithm in Figure 3 takes polynomial time. Suppose the average number of clauses in each conditional expression is $N_c$, then the time complexity of Algorithm 2 is $O(KN^2_c)$. $K$ is the number of variables in the conditional parts, which is 5 for packet filter example. In the worst case, $N_c$ can be equal to the length of a variable’s value dimension.

4 General Case of Actions: Composite-actions

In the packet filter example we have discussed above, an important constraint is that there is only one action in the action part of every rule. But this is not always held for some other RBSs. For example, IP Security Architecture (IPsec) is another typical RBS. In the rule table of IPsec, besides forward and discard, there are many other kinds of actions, e.g., encrypting with Authentication Header (AH), encrypting with encapsulating Security Payload (ESP) and etc. Beside the increase of the action types, another change lies in that there can be multiple actions in the action part of each rule. We call an action part that has multiple actions as a composite-action. The conflict between the single actions of any two composite-actions may result in the conflict between these two composite-actions. So we need an efficient algorithm to check whether two composite-actions conflict. In the rest of this paper, unless we indicate explicitly, the term action means single action.

4.1 Conflict Between Composite-actions

We need to formally define the meaning of conflict between two composite-actions. First we define a relation $R_{\text{conflict}}$ on the set of all possible actions.

**Definition 3. Conflict Relation**

The conflict relation $R_{\text{conflict}}$ on action set $S_A = \{A_1, A_2, \ldots, A_m\}$ is defined as

$$R_{\text{conflict}} = \{(A_i, A_j) : A_i \text{ conflicts with } A_j, A_i, A_j \in S_A\}.$$  \[\Box\]

As mentioned above, a composite-action has multiple actions. Two composite-actions are conflicting if there is at least one pair of actions, each of which comes from a different composite-action and which are conflicting. Definition 4 gives a formal description.
Definition 4. Conflict Composite-actions
For two rules $R_i$ and $R_j$:

\[
R_i : \text{if} (C_i) \text{then} (A_{i1}; A_{i2}; \ldots; A_{im}) \quad \text{and} \\
R_j : \text{if} (C_j) \text{then} (A_{j1}; A_{j2}; \ldots; A_{jm}),
\]

their action parts conflict if and only if there is at least one pair of actions \((A_{ip}, A_{jq}) \in R_{\text{conflict}}\) \((1 \leq p \leq m, 1 \leq q \leq M)\). □

4.2 Checking Conflict Between Composite-actions

Figure 4 shows the algorithm to check conflict between two composite-actions. In this algorithm, we use a graph to represent the conflict relation $R_{\text{conflict}}$. We call this graph the conflict action graph. Every node in this graph stands for an action and there is an edge between two nodes if and only if the corresponding two actions conflict. The time complexity of this algorithm is $O(M^2)$. $M$ is the maximal number of actions.

```
Algorithm 3: Checking conflict composite-actions
Input: Two composed-actions $A_i$ and $A_j$, conflict action graph
Output: Boolean value to show if they are conflict

1 Initialize all entries in array color[M] to white;
2 For every node $a_k$ in the conflict action graph of $R_{\text{conflict}}$, do
3     if $a_k \in A_i - A_j$ then color[k]=blue;
4     if $a_k \in A_i - A_j$ then color[k]=red;
5     if $a_k \in A_j - A_i$ then color[k]=green;
6 For every node $a_k \in A_i$ do
7     For every node $a_k \in A_j$ do
8     if (color[p] ≠ color[q]) or (color[p] == color[q] == green) return true;
9     return false;

Figure 4. Algorithm for checking conflict composite-actions
```

In this algorithm, we use a color array to keep a record for the appearance of each node in the two composite-actions. If a node is in $A_i$, we set its color to blue; if a node is in $A_j$, we set its color to red; and if a node is in both $A_i$ and $A_j$, we set its color to green. After that, every edge related to the two “composite actions” is checked. If the colors of one edge’s two end points are either both green or different, we say these two composite-actions conflict with each other. In Algorithm 3, we treat composite-actions as sets of actions, so set operators can apply [8].

5 General Case of Conditional Expressions

In this section, we discuss the general case of conditional expressions. In Section 3, we assumed that the values of each of the variables used in the conditional expressions of rules are a finite set of integers. For most RBSs, being discrete is still a general attribute of variables, but the other two
attributes, finite and ordered, may not hold any more. In this section we relax our requirement for
the variables and give the definition of conditional expressions in general case using Boolean
Logic.

In Boolean Logic we use Boolean variables $x_1, x_2, \ldots$ for the individual statement such as
“destination port lower than 255”. That is, each Boolean variable denotes a statement that can in
principle be true or false independently of the truth-value of the others [8].

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a finite set of Boolean variables, and let $\overline{X} = \{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$, where the
$\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}$ are new symbols standing for the negations of $x_1, x_2, \ldots, x_n$. We call the elements of
$X \cup \overline{X}$ literals; variables are positive literals, whereas negations of variables are negative literals.
A clause $L$ is a nonempty set of literals: $L \subseteq X \cup \overline{X}$. Finally a Boolean formula in conjunctive
normal form is a set of clauses defined on $X$.

For example, if we have $X = \{x_1, x_2, x_3\}$, and therefore $\overline{X} = \{\overline{x_1}, \overline{x_2}, \overline{x_3}\}$. $L = \{\overline{x_1}, \overline{x_2}, x_3\}$ is a
clause. If we use parentheses instead of the usual set brackets and also separate the literals by the
delimiter or instead of coma, clause $L$ can be rewritten as a disjunctive only Boolean expression
$L = (\overline{x_1} \lor \overline{x_2} \lor x_3)$. So a Boolean formula can be rewritten as a Boolean expression in conjunctive
normal form, interested readers can refer to [8][9] for algorithms and examples.

Because any conditional expression can be transformed into a conjunctive normal form, we define
the conditional expression in general case as:

**Definition 5.** *Condition expression in general case*

The conditional expression of a rule is a Boolean formula in conjunctive normal form.

Because every conditional expression can be represented by a Boolean formula in conjunctive
normal form, the intersection of two conditional expressions is the conjunction of the two
Corresponding Boolean formulas, which is still a Boolean formula in conjunctive normal form. So
the problem of checking whether two generalized condition expressions are intersected equals to
check the satisfiability of a Boolean formula in conjunctive normal form. It takes linear time to
check the satisfiability of a Boolean formula in a normal form.

6 Case study

We implement our algorithms under LINUX and apply it on a set of rules collected from one of the
firewall systems in Bell Labs. Because of security reason, we do not include the source data in this
paper but only report our experimental results.

The rule table we have checked includes a total of 48 rules. They do filtering on various types of IP
traffic, for example, TCP packet of mail service with destination port 25, virus alert on specific IP
address, and even popular ICQ traffic (UDP traffic send to destination port 4000), etc. The fields in
the IP header being checked include *source address, destination address, protocol type, source
port and destination port*.

The difference between this real table and the one mentioned in the previous sections is that the real
rule is attached with another attribute, direction. The direction may be IN, OUT and BOTH, with
the meaning that this rule should be applied on a packet when the packet is received, forwarded or both respectively. Because the IN rules and OUT rules are applied at different time and cannot conflict with each other, we rewrite the original table to two separate tables in accordance with the different directions.

There is a default rule in every table, which is the last rule be applied when all other rules are failed. It is always with the following format,

\[ if ((src = *) and (dst = *) and (ptl = *) and (srcport = *) and (dstport = *)) \text{then} (\text{discard}) \]

Here the star (*) means all possible values. For example, (src = *) means the source IP address can be anyone between 0.0.0.0 and 255.255.255.255. According to our definition of conflict, the last rule will conflict with every rule with the action forward. To simplify our analysis, we ignore this default last rule when applying our algorithms.

In the remaining 47 rules, we have found 43 potential conflicts. We say a pair of rules conflicts when the traffic described by the two rules has overlap (or there is traffic that can satisfy both rules) but the actions taken by the two rules are different. This is consistent with our definition of conflict in Definition 1. Following we analyze two of potential conflicts we found.

1) Rule 4 and 7,

<table>
<thead>
<tr>
<th>No.</th>
<th>Content of rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>If (packets with src address direct_mail and service mail) then (forward it)</td>
</tr>
<tr>
<td>7</td>
<td>If (packets with dst address russtroj) then (discard it)</td>
</tr>
</tbody>
</table>

Here the term mail stands for the mail service that has protocol type TCP and destination port 25. So if a mail packet from the host direct_mail to host russtroj comes, rule 4 tells the packet should be forwarded, while rule 7 states it should be discarded. Because rule 4 is applied before rule 7, the action taken by the firewall will be “forward it”. But is this the desired action by the system administrator?

2) Rule 9 and 14,

<table>
<thead>
<tr>
<th>No.</th>
<th>Content of rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>If (packets with src address mh and service sunrpe) then (discard it)</td>
</tr>
<tr>
<td>14</td>
<td>If (packets with src address bandt2 and service bandt2_out) then (forward it)</td>
</tr>
</tbody>
</table>

The IP addresses group bandt2 is part of mh, sunrpe is the UDP service with destination port 111, and bandt2_out means the service with protocol UDP and source port 7331. So if a packet from bandt2 to somewhere with service udp/7331/111 comes, rule 9 tells the packet should be discarded, while rule 14 says it should be forwarded. Because rule 9 is applied before rule 14, the action taken by the firewall will be “discard it”. But is this the desired action by the system administrator?

As we claimed, our findings are just potential conflicts. Only the system administrator who wrote these rules can decide if they are real conflicts. If the action taken by the firewall system is not what the system administrator has desired, then it’s a real conflict. Otherwise, it is only the real intention of the system administrator.
7 Conclusion

In this paper, we proposed a solution for the fault detection problem for packet filter in firewall systems, which is a typical type of RBS. We also extended our solution to deal with composite-actions. For RBS with generalized conditional expressions, the fault detection problem is NP-hard. We also applied our algorithms on a set of real data and reported the result. Our future work will include two aspects. One is to explore other applications than the packet filter in firewall systems; the other is to search for heuristic algorithms for the general RBS.

Acknowledgements: We would like to thank Eric Grosser for providing us the real data collected from firewalls in Bell Labs and commenting on our experimental results.

References