Journal of Information Processing Vol. 16 142–156 (Aug. 2008)

Regular Paper

Detection of Conflicts Caused by a Combination of Filters Based on Spatial Relationships

Yi Yin,[†1] Yoshiaki Katayama[†1]
and Naohisa Takahashi[†1]

Packet filtering in firewalls is one of the useful techniques for network security. This technique examines network packets and determines whether to accept or deny them based on an ordered set of filters. If conflicts exist in filters of a firewall, for example, one filter is never executed because of the prevention of a preceding filter, the behavior of the firewall might be different from the administrator's intention. For this reason, it is necessary to detect conflicts in a set of filters. Previous researches that focused on detecting conflicts in filters paid considerable attention to conflicts caused by one filter affecting another, but they did not consider conflicts caused by a combination of multiple filters. We developed a method of detecting conflicts caused by a combination of filters affecting another individual filter based on their spatial relationships. We also developed two methods of finding all requisite filter combinations from a given combination of filters that intrinsically cause errors to another filter based on top-down and bottom-up algorithms. We implemented prototype systems to determine how effective the methods we developed were. The experimental results revealed that the detecting conflicts method and the method of finding all requisite filter combinations based on the bottom-up algorithm can be used for practical firewall policies.

1. Introduction

Network security can be increased by filtering packets at firewalls. Packet filtering is a technique by which network packets are accepted or denied based on an ordered set of filters and is called a **firewall policy**. Each filter in a firewall policy consists of some conditions and an action (*accept* or *deny*). When the header fields of an incoming packet satisfy a filter's conditions, this filter's action will be carried out on the incoming packet. A first-matching scheme is used by many packet filtering systems such as *IPFW* in FreeBSD [22] and *Iptables* in Linux [24]. When a packet, P, arrives, the firewall compares P's header fields with the conditions of the filter from the first to the last in turn until it finds one filter f whose conditions satisfy P's header fields. The firewall then uses the action of filter f to deal with packet P.

Consider a firewall policy that contains filters f and g: f precedes g, and f accepts all packets that should be denied by g. In this case, we can say that f causes an **error** to g because f prevents g from denying packets. Of all the packets that match g, g is only executed when none of the arriving packets match f in any way. In addition, g is not executed if the arriving packet completely matches f. In this case, we can say that filter f causes a **warning** to filter g. Such errors and warnings are called **conflicts** [3),5),6)]. It is even difficult for experienced administrators to detect conflicts that occasionally occur in firewall policies due to their modification. Since conflicts are difficult to detect, they may easily occur in firewall policies. In examining 37 firewalls in production enterprise networks in 2004, Wool found that all the network's firewalls were mis-configured and vulnerable [1)]. Various techniques have been developed to help network administrators detect conflicts in firewall policies [2)–6)]. Although these techniques can help administrators detect conflicts, they are limited because they only consider conflicts caused by one filter to another filter.

Consider a firewall policy that contains filters f_1, f_2, f_3 and g: f_1, f_2, and f_3 precede filter g, and we want to check whether f_1, f_2, and f_3 cause conflicts to g. We assumed filter f_1 would *accept* all the packets with a destination port number ranging from 0 to 45, f_2 would *accept* all the packets with a destination port number ranging from 23 to 80, f_3 would *accept* all the packets with a destination port number ranging from 80 to 100, and g would *deny* all the packets with a destination port number ranging from 10 to 60. If we only consider conflicts caused by one filter to another, f_1 only causes a warning and does not cause any error to g, and f_2 also only causes a warning and does not cause any error to g. However, because any of the packets that match g also match f_1 and f_2, g is never executed. In this case, we can say that a **combination** of filters f_1 and f_2 causes an error to filter g. Therefore, it is necessary to detect conflicts caused by a combination of filters in a firewall policy.

In the firewall policy mentioned above, some combinations of filters that may

†1 Nagoya Institute of Technology

cause conflicts to filter g can be considered, such as a combination of filters f_2 and f_3, a combination of filters f_1, f_2 and f_3, and so on. If we consider the combination of filters f_1, f_2 and f_3, we can determine that it also causes an error to filter g, but we also know that the single filter f_3 does not cause any conflict to filter g. Therefore, it is important to remove irrelevant filters, such as filter f_3, from a combination that causes an error to filter g, such as the combination of filters f_1, f_2 and f_3, because the combination of filters f_1 and f_2 intrinsically causes error to filter g while filter f_3 does not essentially contribute to causing an error to filter g. We call filters f_1 and f_2 **a requisite filters combination for an error** to filter g. In general, a combination of K filters ($K \geq 2$) is called a **requisite filters combination for an error** to another filter when an error is caused by this combination, and no errors (i.e. warnings or no conflicts) are caused by any of the combinations of $(K-1)$-filters that do not contain a filter from the combination of K filters. An individual filter f is called **a requisite filter for an error** to another filter when an error is caused by this filter. **A requisite filters combination for an error** and **a requisite filter for an error** are represented as ***RFC***.

We developed a **conflict detection method**, ***CDM***, and two ***RFC* finding methods**: the ***RFM-T*** and ***RFM-B***. Given a combination of K filters, and a filter g, where the K filters are placed before filter g, the CDM detects conflicts caused by the combination of K filters to g using their spatial relationships. For the combination of K filters and filter g, RFM-T identifies all RFCs from the combination of K filters based on a top-down algorithm, while RFM-B identifies all RFCs from the combination of K filters based on a bottom-up algorithm.

The rest of this paper is organized as follows: Section 2 introduces the background of firewalls. Section 3 introduces related works. Section 4 describes the conflicts caused by combinations of filters. Section 5, describes the implementation of methods that have been developed. Section 6, compares the two algorithms on the basis of experiments. Finally, we conclude the paper.

2. Background

A firewall policy, FP, consists of an ordered set of n filters, and it is expressed as follows:

FP : (f_1, f_2, ... , f_n),

where if two filters, f_i and f_j ($i, j \in [1, n]$ and $i \neq j$), satisfy $i < j$, filter f_i is placed before f_j in FP. Each filter, f_i, ($i \in [1, n]$) consists of m **predicates**, p_{i1}, p_{i2}, ..., p_{im}, and an action, and f_i is expressed as follows:

f_i: p_{i1}, p_{i2}, ... , p_{im}, action

where m is the number of header fields to be used in packet filtering. The commonly used header fields are: protocol type, source IP addresses (represented as **SrcIP**), destination IP addresses (represented as **DesIP**), source port number (represented as **SrcPort**) and destination port number (represented as **DesPort**).

Each **predicate** p_{ij} ($i \in [1, n]$, $j \in [1, m]$) in a filter, is a matching condition for a packet header field, and it commonly allows four kinds of matches: *exact match, prefix match, range match*[18], *or list match*[23]. In an exact match, the packet header field should exactly match the predicate; this is useful for specifying the protocol, for instance. In a prefix match, the predicate should be a prefix of a packet header field; this could be useful for blocking access from a certain subnetwork. In a range match, the header values of a packet should lie in the range specified by the predicate; this can be useful for specifying port number ranges. In a list match, the header value of a packet should be one of the items within the list; for instance, this is useful for specifying multiple similar predicates within a filter, such as specifying multiple discrete port numbers in a filter.

As previously stated the action of a filter is either "accept" or "deny".

We used a first-matching scheme, which is used by many systems, such as $IPFW$ in FreeBSD[22], and the meaning of first-matching was explained in Section 1. The default filter of a firewall policy is a filter that can match packets when no other filters of the firewall policy can, and it is always the last filter in a firewall policy. In this paper, we used *deny all* as the default filter that denies all packets.

A filter where each predicate is represented as an exact value, a prefix, a range value or a list is called an **external form filter**. In our work, each predicate p_{ij} ($i \in [1, n]$, $j \in [1, m]$) in a filter is represented as a uniform range value, $[a_{ij}, b_{ij})$. A filter where each predicate is represented as a uniform range value is called an **internal form filter**. An internal form filter, f_i ($i \in [1, n]$), is represented as

	SrcIP	DesIP	Action
f_1:	[1, 6),	[1, 6),	accept
f_2:	[2, 7),	[4, 9),	accept
g_1:	[3, 5),	[2, 7),	deny
g_2:	[0, 2^{32}),	[0,2^{32}),	deny

Fig. 1 Example firewall policy.

	DesPort	Action
f_1:	[0, 46),	accept
f_2:	[23, 81),	accept
f_3:	[81, 100),	accept
g_1:	[10, 61),	deny

Fig. 2 Example filters 1.

follows:

f_i: $[a_{i1}, b_{i1})$, $[a_{i2}, b_{i2})$, ..., $[a_{im}, b_{im})$, action

Assume the values of the header fields of packet P are $(x_1, x_2, ..., x_m)$, if and only if $a_{i1} \leq x_1 < b_{i1}$, $a_{i2} \leq x_2 < b_{i2}$, ..., $a_{im} \leq x_m < b_{im}$, packet P matches filter f_i, and the action of filter f_i is performed on packet P. An example firewall policy consists of internal form filters, as shown in **Fig. 1**, where filter g_2 represents a default filter.

External form filters can easily be converted into internal form filters. For example, if an exact value predicate of 140.192.37.60 is given, we can represent this as [140.192.37.60, 140.192.37.61), if a prefix predicate of 140.192.37.* is given, we can represent this as [140.192.37.0, 140.192.38.0), if a list predicate {192.168.0.1, 10.5.32.6} is given, we can split this into two exact value predicates and then represent each of them in the internal form. We have omitted the method that is used to transform external form filters into internal form filters because transformation is outside the scope of this paper.

3. Related Work

Over the past few years, researches on the analysis of firewall policies have received broad attention [2)–17)], and they can be roughly classified into three main groups: (1) detection of conflicts in firewall policies [2)–6),15)–17)], (2) removal of redundant filters in firewall policies [7),8)], and (3) interactive analyzers of firewall policies [9)–14)].

(1) Detection of Conflicts in Firewall Policies.

Al-Share, et al. proposed an algorithm to detect conflicts caused by one filter to another in a firewall policy based on the relations between every two filters [2),3)]. The relations between the two filters were defined based on their predicates as to whether they could satisfy the condition in $\{ \supset, \subset, = \}$. However, their research has two problems. The first problem is that when the corresponding predicates of two filters overlap, such as the filters in **Fig. 2**, the relations between the two filters cannot be determined because overlapping predicates do not satisfy any of the conditions in $\{ \supset, \subset, = \}$. Therefore, the algorithm for detecting conflicts also does not work when overlapping predicates appear between two filters. The second problem is that they cannot detect conflicts caused by a combination of filters to another filter because they cannot determine the relations between two filters when overlapping predicates appear in them.

In order to avoid overlapping filters and to find conflicts between every two filters in a firewall policy, Al-Share, et al. break down two filters with the overlapping predicates into several filters which are equivalent to the two filters as a whole [3)]. Their problems were that they did not provide a way of breaking down the overlapping predicates of two filters and that they only considered conflicts caused by one filter to another. Because a conflict caused by a combination of filters may be divided into some conflicts between two filters, the way overlapping predicates are broken down determines whether conflict caused by a combination of filters can be detected. For example, if filters f_1 and f_2 as shown in Fig. 2 are broken into equivalent filters f_{11}, f_{12}, f_{13}, and f_{21}, f_{22} as shown in **Fig. 3**, where the predicates of each of the two filters satisfy one of the conditions in $\{ \supset, \subset, = \}$, because they only considered conflicts caused by one filter to another, they could not detect conflicts such as filter g_1 is never executed by the combination of filters f_{11}, f_{12}, f_{13} and f_{21}, f_{22}. Therefore, although their conflict detection algorithm for two filters can be used after overlapping predicates are broken down, Al-Share, et al. also occasionally could not detect conflicts to another filter caused by a combination of filters.

Yuan, et al. presented a static analysis algorithm to check the misconfigurations of an individual firewall as well as distributed firewalls [4)]. They implemented their algorithm in a tool *FIREMAN* based on a binary decision diagrams (BDDs). Although they mentioned a part of the conflicts that are presented in this paper, their goal was to achieve fast misconfigurations detection of a single firewall and of distributed firewalls, but not to detect all the conflicts caused by a combination of multiple filters and to decide whether a combination of filters is an *RFC*

or not. Baboescu, et al. addressed the problems of fast updates and fast conflict detection, and they proposed an efficient and scalable conflict detection algorithm [5]. Although this research could be used for large scale firewall policies, it only considered the conflicts caused by one filter to another, therefore, it cannot be used to find the conflicts caused by a combination of filters and it cannot detect all $RFCs$. Hari, et al. proposed algorithms for detecting and resolving conflicts in a filter database [6]. Although this research can be used to detect and resolve the conflicts, it also only considered the conflicts caused by one filter to another. Therefore, it cannot be used to detect conflicts caused by a combination of filters and detect all $RFCs$.

In our previous works [15)–17)], we discussed the problem of the possible effect to another filter caused by adding a filter in a firewall policy. This kind of effect is called **side effect** and the analysis method of this problem is called **side effect analysis**. This paper improves and generalizes the side effect analysis presented in Refs. 15)–17), to detect conflicts caused by a combination of filters to another filter by using their spatial relationships. In terms of the conflicts of firewall policies, several other researches expressed it differently, such as *errors* in Ref. 1) or *misconfigurations* in Ref. 4) and so on.

(2) Removal of Redundant Filters in Firewall Policies.

The literature [7),8)] presents methods to reduce the number of filters without changing the meaning of a firewall policy by removing the redundant filters. Liu, et al. used a firewall decision tree (FDTs) to detect and remove all the redundant filters [7]. K. Matsuda proposed a model called "matrix decomposition" which enables to analyze filters, and some firewall policy compression methods (removable filters deletion, verbose filters revision and filters combination) by using this model [8].

The goal of our research is to detect the conflicts caused by a combination of filters which are not easily detectable by an administrator and show them to the administrator. To accomplish this goal, we propose a conflict detection method (CDM) and two RFC finding methods (RFM-T and RFM-B) which are based on the spatial relationships between the filters.

For some reasons, an administrator can intentionally embed redundant filters, which will conflict with the other filters in a firewall policy. In this case, we will inform the administrator of the fact that there exist some conflicts in the firewall policy instead of removing these redundant filters. Furthermore, to help the administrator decide of what should be done (adding, deleting, or replacing filters to deal with conflicts) according to his intentions, we classify the conflicts and show him the contents of conflicts in detail. For the reason that we want to inform the administrator about the conflicts, it is necessary to try getting rid of filters that do not essentially contribute to causing conflicts. Therefore, we proposed RFC finding methods (RFM-T and RFM-B) to reduce the number of filters that have to be checked by the administrator.

(3) Interactive Analyzer of Firewall Policy.

Hazelhurst, et al. described a method of transforming a firewall policy written in a Cisco-like access list language into a BDD (binary decision diagrams) [9),10)]. Their goal is to achieve fast lookup by using BDDs. Although this method can be used to answer questions about the types of packets permitted or excluded by a set of filters and even to find redundant filters of a firewall policy, it cannot be used to find conflicts caused by a combination of filters to another filter and all RFCs. Mayer, et al. developed a firewall analysis tool, $Fang$, to perform customized queries on a set of filters and to extract the filters in a firewall policy [11),12)]. Wool, et al. improved the usability of *Fang* by automating the queries to check whether firewalls are configured according to the administrator's expectations or not [13)]. Eronen and Zitting presented an expert system based on Eclipse to verify the functionality of filters by performing queries [14)].

All these tools and methods help the administrator manually verify the correctness of a firewall policy. Unfortunately, they require a high degree of user expertise to write proper queries to identify different firewall policies' problems, and they cannot be used to find conflicts caused by a combination of filters to another filter.

4. Classification of Conflicts

4.1 Packet Space and Filter Space

When the number of header fields in a packet is m, the packet can be represented as a point in an m-dimensional space called a **packet space**. A filter is represented as a sub-space of a packet space called a **filter space**, which includes

	DesPort	Action
f_{11}:	[0, 10),	accept
f_{12}:	[10, 23),	accept
f_{13}:	[23, 46),	accept
f_{21}:	[23, 61),	accept
f_{22}:	[61, 81),	accept
f_3:	[81, 100),	accept
g_1:	[10, 61),	deny

Fig. 3 Example filters 2.

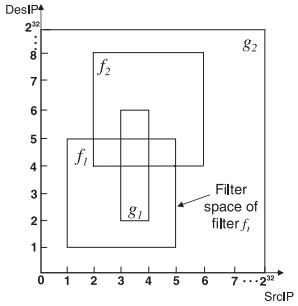


Fig. 4 Filter spaces in packet space.

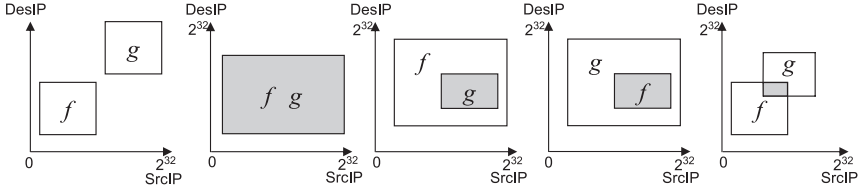
all the points of packets that match the filter. The filter space of a filter, f, is represented as $\mathscr{S}(\boldsymbol{f})$. For example, all filters in the firewall policy of Fig. 1 contain two predicates for two header fields, i.e., "SrcIP" and "DesIP" of packets. The filter spaces of f_1, f_2, and g_1 are represented as rectangles in a two-dimensional packet space, and the filter space of g_2 is the whole packet space, as we can see in **Fig. 4**.

4.2 Spatial Relationships of Filters

The relationships between the filter spaces of filters f and g are called the **spatial relationships between filters $\boldsymbol{f}$ and $\boldsymbol{g}$** and are represented as $\mathscr{R}(\boldsymbol{f}, \boldsymbol{g})$, where $\mathscr{R}(f, g) \in \{$*Disjoint, Equivalent, Inclusion*1, *Inclusion*2, *Correlation*$\}$. $\mathscr{R}(f, g)$ is determined by the filter spaces of f and g, i.e., $\mathscr{S}(f)$ and $\mathscr{S}(g)$, and is expressed as follows:

$$\mathscr{R}(f,\ g) = \begin{cases} \mathit{Disjoint} & \text{if } \mathscr{S}(f) \cap \mathscr{S}(g) = \emptyset \\ \mathit{Equivalent} & \text{if } \mathscr{S}(f) = \mathscr{S}(g) \\ \mathit{Inclusion}1 & \text{if } \mathscr{S}(f) \supset \mathscr{S}(g) \\ \mathit{Inclusion}2 & \text{if } \mathscr{S}(f) \subset \mathscr{S}(g) \\ \mathit{Correlation} & \text{otherwise} \end{cases} \tag{1}$$

Fig. 5 (a)–Fig. 5 (e) show the images of the spatial relationships of {*Disjoint, Equivalent, Inclusion*1, *Inclusion*2, *Correlation*} respectively. The hatching parts of Fig. 5 (a)–Fig. 5 (e) show the common space of two filter spaces $\mathscr{S}(f)$ and $\mathscr{S}(g)$.



$\mathscr{R}(f, g)$=*Disjoint* Fig. 5 (a) | $\mathscr{R}(f, g)$=*Equivalent* Fig. 5 (b) | $\mathscr{R}(f, g)$=*Inclusion*1 Fig. 5 (c) | $\mathscr{R}(f, g)$=*Inclusion*2 Fig. 5 (d) | $\mathscr{R}(f, g)$=*Correlation* Fig. 5 (e)

Fig. 5 Spatial relationships of filters f and g.

4.3 Combination of Filters

Let us introduce a notation, $\mathbf{C}[\boldsymbol{f}_1, \boldsymbol{f}_2, ..., \boldsymbol{f}_K]$, to represent **a combination of $\boldsymbol{K}$ filters**, or a **$\boldsymbol{K}$-filters combination**, where the K filters, $f_1 \sim f_K$, all have the same actions, and an individual filter f also can be seen as a combination of K filters when K is equal to one. For simplicity, we use a symbol, $\mathbf{C}^K$, to represent a combination of K filters, $\mathrm{C}[f_1, f_2, ..., f_K]$. We also define $\mathbf{C}^K$**.action** to represent the action of a combination of K filters, and $\boldsymbol{g}$**.action** to represent the action of filter g. We define $\mathscr{S}'(\mathbf{C}^K)$ and $\mathscr{R}'(\mathbf{C}^K, \boldsymbol{g})$ to denote the **$\boldsymbol{K}$-filters combination space** and the **spatial relationships between the $\boldsymbol{K}$-filters combination and a filter, $\boldsymbol{g}$**, as follows:

$$\mathscr{S}'(C^K) = \mathscr{S}(f_1) \cup \mathscr{S}(f_2) \cup ... \cup \mathscr{S}(f_K). \qquad (K \geq 2) \tag{2}$$

$$\mathscr{S}'(\mathrm{C}[f]) = \mathscr{S}(f), \qquad f \in [f_1, ...f_K], \qquad (K = 1) \tag{3}$$

$$\mathscr{R}'(C^K,\ g) = \begin{cases} \mathit{Disjoint} & \text{if } \mathscr{S}'(C^K) \cap \mathscr{S}(g) = \emptyset \\ \mathit{Equivalent} & \text{if } \mathscr{S}'(C^K) = \mathscr{S}(g) \\ \mathit{Inclusion}1 & \text{if } \mathscr{S}'(C^K) \supset \mathscr{S}(g) \\ \mathit{Inclusion}2 & \text{if } \mathscr{S}'(C^K) \subset \mathscr{S}(g) \\ \mathit{Correlation} & \text{otherwise} \end{cases} \qquad (K \geq 2) \tag{4}$$

$$\mathscr{R}'(\mathrm{C}[f], g) = \mathscr{R}(f, g), \qquad f \in [f_1, ...f_K], \qquad (K = 1) \tag{5}$$

4.4 Classification of Conflicts

We assume that a firewall policy consists of an ordered set of n filters, $f_1 \sim f_n$, and a filter g, where if two filters, f_i and f_j ($i, j \in [1, \mathrm{n}]$, and $i \neq j$), satisfy $i < j$, f_i is placed before f_j, and the n filters are placed before the filter g. If we take out K

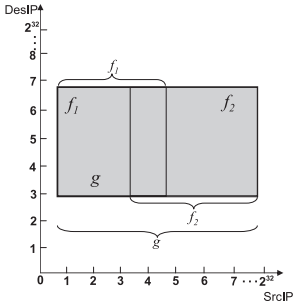


Fig. 6 $\mathscr{R}$'(C[f_1, f_2], g)=*Equivalent*.

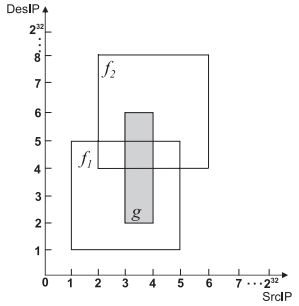


Fig. 7 $\mathscr{R}$'(C[f_1, f_2], g)=*Inclusion*1.

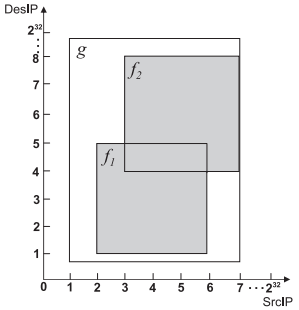


Fig. 8 $\mathscr{R}$'(C[f_1, f_2], g)=*Inclusion*2.

filters, $f_1 \sim f_K$, that have the same actions from the n filters, the possible conflicts caused by C^K to filter g can be classified as follows: These conflicts caused by a combination of K filters were inspired by the classification of conflicts between two different filters [2),3)].

a). Shadowing error: When filter g is never executed because C^K prevents g from accepting or denying packets, we can say that filter g is **shadowed** by C^K and there exists a **shadowing error** between C^K and g.

Figure 6 shows an example where the $\mathscr{R}$'(C[f_1,f_2], g) is *Equivalent*, and **Fig. 7** shows an example where $\mathscr{R}$'(C[f_1,f_2], g) is *Inclusion*1. The hatching parts of Fig. 6 and Fig. 7 are the common spaces of $\mathscr{S}$'(C[f_1,f_2]) and $\mathscr{S}(g)$. We assume the action of C[f_1,f_2] is "*accept*" while the action of filter g is "*deny*". For the reason that all the packets satisfying filter g also satisfy C[f_1,f_2], all these packets that should be denied by filter g will be accepted by C[f_1,f_2]. Therefore, the action of filter g is never executed.

We generalized C[f_1,f_2] in Fig. 6 and Fig. 7 to C^K and summarized the conditions under which a shadowing error occurs as follows. The shadowing error occurs between C^K and filter g when any one of the following conditions is true.
(1). $\mathscr{R}$'(C^K, g) = *Equivalent*, and C^K.action $\neq$ g.action.
(2). $\mathscr{R}$'(C^K, g) = *Inclusion*1, and C^K.action $\neq$ g.action.

b). Redundancy error: When C^K accepts or denies the same packets that filter g wants to accept or deny, such that if filter g is removed from the firewall policy, the firewall policy will not be affected. At this time, we can say that g is a **redundant filter** to C^K and there exists a **redundancy error** between C^K and g. A redundant filter will unnecessarily increase the size of the filter list and, therefore, increase the search time and space requirements for filtering packets.

We use the same example shown in Fig. 6 and Fig. 7, but we assume the actions of C[f_1,f_2] and filter g are both as "*accept*". At this time, the packets to be accepted by filter g have already been accepted by C[f_1,f_2]. Therefore, if filter g is removed, the packets to be accepted by filter g also can be accepted by C[f_1,f_2].

We generalized C[f_1,f_2] in Fig. 6 and Fig. 7 to C^K and summarized the conditions under which a redundancy error occurs as follows. The redundancy error occurs between C^K and filter g when any one of the following conditions is true.
(1). $\mathscr{R}$'(C^K, g) = *Equivalent*, and C^K.action = g.action.
(2). $\mathscr{R}$'(C^K, g) = *Inclusion*1, and C^K.action = g.action.

c). Generalization warning: Filter g is a **generalization filter** to C^K and there exists a **generalization warning** between C^K and filter g when the following condition is true.
(1). $\mathscr{R}$'(C^K, g) = *Inclusion*2, and C^K.action $\neq$ g.action.

If a generalization warning occurs, the specific combination of K filters, C^K, generates an exception for the general filter, g, and filter g is only executed when a certain range of packets arrive.

For example, **Fig. 8** shows an example where the $\mathscr{R}$'(C[f_1, f_2], g) is *Inclusion*2. The hatching part of Fig. 8 is the common space of $\mathscr{S}$'(C[f_1,f_2]) and $\mathscr{S}(g)$. If the actions of C[f_1, f_2] and filter g are different, filter g is a generalization filter

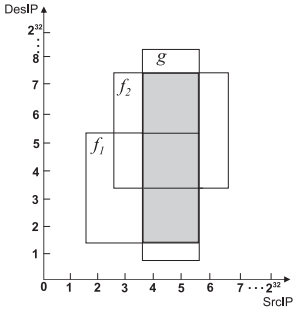


Fig. 9 $\mathscr{R}$'(C[f_1, f_2], g)=*Correlation*.

to C[f_1, f_2], and a generalization warning occurs between C[f_1, f_2] and filter g.

d). Correlation warning: Filter g is a **correlation filter** to C^K and there exists a **correlation warning** between C^K and filter g when the following condition is true.

(1). $\mathscr{R}$'(C^K, g) = *Correlation*, and C^K.action $\neq$ g.action.

If a correlation warning occurs, C^K and filter g imply an action that is not explicitly stated, and filter g is only executed when a certain range of packets arrive.

For example, **Fig. 9** shows an example where the $\mathscr{R}$'(C[f_1, f_2], g) is *Correlation*. The hatching part of Fig. 9 is the common space of $\mathscr{S}$'(C[f_1,f_2]) and $\mathscr{S}(g)$. If the actions of C[f_1, f_2] and filter g are different, filter g is a correlation filter to C[f_1, f_2], and a correlation warning occurs between C[f_1, f_2] and filter g.

The spatial relationship between the combination of K filters and filter g, i.e., $\mathscr{R}$'(C^K, g), is the important factor in deciding whether C^K will cause conflicts to filter g. How filters are represented in a spatial space and how spatial relationships are calculated between the combination of K filters and filter g are described in Sections 5.3 and 5.4.

5. Detecting Conflicts and Finding Combination of Requisite Filters

We present a conflict detection method (CDM) and two RFC finding methods (the RFM-T and RFM-B), with the same combination of K filters C^K and the filter g described in Section 4.4.

5.1 Conflict Detection Method

The CDM (conflict detection method) uses a function $DecideConflict()$, which receives C^K and g as input, and outputs the kind of conflict between C^K and g. The complete algorithm is given in **Fig. 10**. In the algorithm, let "*CK*" be a filter set to represent a combination of K filters, and "*Conflict*" be the kind of conflict between C^K and g.

The $DecideConflict()$ firstly checks the spatial relationships between the combination of K filters and filter g, i.e., $\mathscr{R}$'(C^K, g). If $\mathscr{R}$'(C^K, g) is *Disjoint*, "*No Conflict*" is stored in the variable "$Conflict$". Finally, the $DecideConflict()$ returns the "$Conflict$". If $\mathscr{R}$'(C^K, g) is one of the {*Inclusion*1, *Inclusion*2, *Generalization, Correlation*}, the function decides the kind of conflict between C^K and g according to the actions of C^K and g, and returns it as output.

5.2 *RFC* Finding Methods

When C^K causes a shadowing error or redundancy error to filter g, in order to identify whether C^K is an RFC to filter g, we introduce two RFC finding methods, i.e., RFM-T and RFM-B. If C^K is not an RFC, the RFM-T and RFM-B also can detect all RFCs that are made from the filters in C^K. The RFM-T is a top-down algorithm while RFM-B is a bottom-up algorithm.

5.2.1 *RFM-T*

The RFM-T identifies whether C^K is an RFC and determines all RFCs. The complete top-down algorithm is given in **Fig. 11**, and the meanings of variables in the algorithm are defined in **Table 1**.

RFM-T has two functions: *TopDown*() and *TopDown1*(). When a combination of K filters C^K and filter g are given as input, *TopDown*(C^K, g) is executed firstly. The function *TopDown*(C^K, g) obtains "*RFCS*" and "*count*" by execution of a recursive function *TopDown1*(C^K, g, *RFCS, count*) (line 4), and outputs them (line 5).

The recursive function *TopDown1*(C^K, g, *RFCS, count*) adds the C^K to *RFCS* in case K=1, because C^K is an RFC (line 16). Then the function returns to the caller *TopDown*(C^K, g) (line 17).

In case $K \geq 2$, *TopDown1*(C^K, g, *RFCS, count*) makes K combinations of $(K-1)$ filters by removing any one filter in C^K, so that each combination of

```
Input:  Combination C[f1, f2, ... , fK],  Filter g
Output: Conflict
Algorithm DecideConflict (C[f1, f2, ... , fK],  g)
 1:  CK=C[f1, f2, ... , fK];
 2:  switch( ℜ' (CK, g) ) {
 3:      case Disjoint: {
 4:          Conflict ← No conflict;
 5:          break;
 6:      }
 7:      case Equivalent:
 8:          if (CK.action=g.action) then {
 9:              Conflict ← Redundancy error;
10:              break;
11:          }
12:          else {
13:              Conflict ← Shadowing error;
14:              break;
15:          }
16:      case Inclusion1:
17:          if (CK.action=g.action) then {
18:              Conflict ← Redundancy error;
19:              break;
20:          }
21:          else {
22:              Conflict ← Shadowing error;
23:              break;
24:          }
25:      case Inclusion2:
26:          if (CK.action=g.action) then {
27:              Conflict ← No conflict;
28:              break;
29:          }
30:          else {
31:              Conflict ← Generalization warning;
32:              break;
33:          }
34:      case Correlation:
35:          if (CK.action=g.action) then {
36:              Conflict ← No conflict;
37:              break;
38:          }
39:          else {
40:              Conflict ← Correlation warning;
41:              break;
42:          }
43:  }
44:  return Conflict;
End of DecideConflict Algorithm
```

Fig. 10 *CDM*: Conflict detection method.

```
Input:  Combination C[f1, f2, ... , fK],  Filter g
Output: RFCS ,  count
Algorithm TopDown ( C[f1, f2, ... , fK] ,  g )
1:    Initialization {
2:        RFCS={ Φ };   count=0;
3:    }
4:    TopDown1(C[f1, f2,..., fK],  g,  RFCS,  count);
5:    Output  RFCS,  count;
End of TopDown Algorithm

Algorithm  TopDown1 (C[f1, f2, ... , fK],  g,  RFCS,  count )
 1:    Initialization {
 2:        flag=0;   newc={ Φ };
 3:    }
 4:    for each filter fi in { f1, f2, ... , fK } do {
 5:        if (K≥2) then {
 6:            newc= the combination of K-1 filters
                    after take out fi from C[f1, f2, ... , fK];
 7:            Conflict=DecideConflict ( newc,  g );
 8:            count ← count+1;
 9:            if ( (Conflict=Redundancy error) or
                   (Conflict=Shadowing error) ) then {
10:                flag=1;
11:                TopDown1(newc, g, RFCS, count );
12:            }
13:        }
14:    }
15:    if (flag=0) then
16:        Add C[f1, f2, ... , fK] to RFCS;
17:    return;
End of TopDown1 Algorithm
```

Fig. 11 *RFM-T*: Top-down algorithm for detecting *RFC*s.

$(K-1)$ filters constitutes a different filter set (lines 4–6). For each combination of $(K-1)$ filters, C^{K-1}, *DecideConflict*() is executed to set the kind of the conflict between C^{K-1} and g into the variable "*Conflict*" (line 7).

If the kind of conflict caused by a combination of $(K-1)$ filters, C^{K-1}, is "Shadowing errors" or "Redundancy errors", "*flag*" is set to be "1" (lines 9–10)

Table 1 Variables in *RFM-T*.

Name	Data Tpye	Meaning
RFCS	Set of combinations	To save all the detected RFCs
count	Integer	To save the number of combinations that should be detected to find all RFCs
flag	Boolean	To mark whether a combination causes an error to filter g
newc	Set of filters	To represent a combination of filters
Conflict	Conflict	To save the result of DecideConflict() function

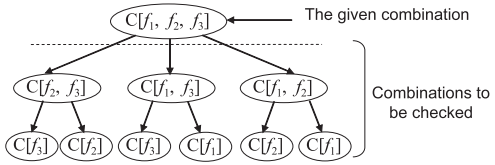


Fig. 12 All the combinations should be checked in the worst case of *RFM-T*.

to note that C^K is not an *RFC*, where the "*flag*" is initialized to be "0" at the beginning of *TopDown1*(C^K, g, *RFCS, count*) (line 2). Then, *TopDown1*(C^{K-1}, g, *RFCS, count*) is called to check whether C^{K-1} is an *RFC* (line 11).

If the "*flag*" holds the initialized value "0" at the end of *TopDown1*(C^K, g, *RFCS, count*), it means that all the K combinations of $(K-1)$ filters do not cause errors to filter g, and C^K is determined to be an *RFC*, then C^K is added to *RFCS* (lines 15–16). At last, *TopDown1*(C^K, g, *RFCS, count*) returns to the caller *TopDown*(C^K, g) (line 17).

When each individual filter, f_i $(i \in [1, K])$, in a combination C^K causes shadowing or redundancy error to filter g, the *RFM-T* will check all possible combinations produced by filters $f_1 \sim f_K$ to find all *RFC*s. Such C^K is in **the worst case** of *RFM-T*. For example, when a combination of three filters C[f_1, f_2, f_3] and filter g are given, where each individual filter f_i $(i \in [1, 3])$ causes shadowing or redundancy error to filter g, to find all *RFC*s, the combinations that should be checked (i.e., all the combinations generated by line 6 in the *TopDown*1 function) are shown as in **Fig. 12**. The checking sequence of them is the same as in depth-first search [21]:

When the given combination C^K is an *RFC* to filter g, the top-down algorithm

Table 2 Variables in *RFM-B*.

Name	Data Tpye	Meaning
RFCS	Set of combinations	To save all the detected RFCs
count	Integer	To save the number of combinations that should be detected to find all RFCs
newc	Set of filters	To represent a combination of filters
Conflict	Conflict	To save the result of DecideConflict() function
T[i] ($i \in [1, K]$)	Temporary combination sets	To save temporary combinations of i filters

only needs to check K combinations of $(K-1)$ filters to determine whether C^K is an *RFC* or not. Such a combination of K filters C^K is in **the best case** of *RFM-T*.

5.2.2 *RFM-B*

The *RFM-B* identifies whether C^K is an *RFC* and determines all *RFC*s in a bottom-up fashion. The complete algorithm is given in **Fig. 14**, and the meanings of variables used in the algorithm are defined in **Table 2**. "*T[i]*" ($i \in [1, K]$) is maintained to store a set of combinations of i filters. For example, "*T[3]*" is a set of combinations of three filters like the following:

T[3]={ C[f_1, f_2, f_3], C[f_2, f_3, f_4], ... }.

RFM-B consists of two parts: lines 6–14, and lines 15–25 in the list of the algorithm in Fig. 14.

The first part splits the C^K into K combinations of one filter, C[f_i] ($i \in [1, K]$) at first (line 6). Then, it determines whether each C[f_i] ($i \in [1, K]$) causes an error to filter g (line 8). If any of the C[f_i] ($i \in [1, K]$) causes a shadowing or a redundancy error to filter g, it will be directly added to "*RFCS*" because C[f_i] ($i \in [1, K]$) is an *RFC* to filter g (lines 10–11). If any of the C[f_i] ($i \in [1, K]$) causes a generalization or a correlation warning to filter g, C[f_i] is added to *T[1]* (lines 12–13). After checking each C[f_i] ($i \in [1, K]$), the first part finishes. Each element in *T[1]* is a combination of one filter which causes a warning to filter g.

When *T[1]* is not empty, and each combination in *T[1]* is added to a filter, some combinations of two filters may cause errors to filter g. The second part checks, when *T[s-1]* is not empty ($s \in [2, K]$) and after adding a filter to each combination in *T[s-1]*, whether new combinations of s filters cause errors to filter g (line 18). If a new combination of s filters causes an error to filter g, it will be added to *RFCS* because it is an *RFC* (lines 21–22). If it causes a warning to filter g, it will be added to *T[s]* (lines 23–24).

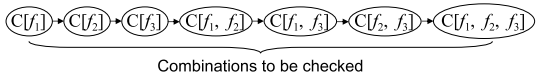


Fig. 13 All the combinations should be checked in the worst case of *RFM-B*.

When each individual filter in C^K causes a generalization or a correlation warning to filter g, and no other combinations produced by filters $f_1 \sim f_K$ cause any shadowing or any redundancy errors to filter g in addition to C^K, the *RFM-B* must check all possible combinations produced by filters $f_1 \sim f_K$ to find all *RFC*s. Such C^K is in **the worst case** of *RFM-B*. For example, when a combination of three filters, C[f_1, f_2, f_3], and filter g are given, all the combinations that should be detected to find all *RFC*s (i.e. all the combinations generated by line 6 and line 18 in the algorithm) are shown as in **Fig. 13**. And the checking sequence is from the first combination C[f_1] to the last combination C[f_1, f_2, f_3].

When each individual filter in a combination of K filters C^K causes an error to filter g, since the *RFC*s are each an individual filter, the *RFM-B* only needs to check K combinations of one filter, i.e., C[f_i] ($i \in [1, K]$), to determine all *RFC*s, and such a combination C^K is in **the best case** of *RFM-B*.

5.3 Calculation of Spatial Relationships between Filters

In Section 4.4, we introduced that conflicts caused by a combination of K filters C^K to filter g are determined by their spatial relationships, i.e., $\mathscr{R}'(C^K, g)$ and their actions. This section outlines how we calculate the $\mathscr{R}'(C^K, g)$.

5.3.1 Cell

First, let us introduce the notation of a **cell**. A cell is a sub-space in a packet space, and has the following characteristics:

(1) Disjoint: No pairs of cells have a common sub-space.

(2) Direct sum: The union of all cells equals the original packet space.

(3) Unitary range: A cell is a sub-space that can be represented by unitary values that range from the first to the last dimension of a packet space.

5.3.2 Steps to Calculate Spatial Relationships between Filters

The shape and the number of cells depend on the method used to divide the

packet space. Our method to divide the packet space is based on SIERRA [16),17)], which is a Systolic filter Sieve Array used in a high-speed packet classifier [18)]. The following steps introduce how $\mathscr{R}'(C^K, g)$ is calculated when C^K and filter g are given.

Step 1: Represent K filters, $f_1 \sim f_K$, and filter g as filter spaces $\mathscr{S}(f_1) \sim \mathscr{S}(f_K)$ and $\mathscr{S}(g)$.

Step 2: Divide the whole packet space into disjoint sub-spaces in the first dimension at all the boundaries of all the filter spaces.

Step 3: Divide each sub-space obtained from the preceding dimensional division in the next dimension at all the boundaries of all the filter spaces.

Step 4: Repeat step 3 until the last dimension of the packet space is reached. Each divided sub-space of a packet space is a cell.

Step 5: Represent each filter space by a set of cells, and represent $\mathscr{S}'(C^K)$ by the union of $\mathscr{S}(f_i)$ (i=1 to K).

Step 6: Calculate $\mathscr{R}'(C^K, g)$ by using the inclusion relationships of the cell set of $\mathscr{S}'(C^K)$ and the cell set of $\mathscr{S}(g)$.

For simplicity, we used the firewall policy in Fig. 1. We also wanted to check whether a combination of filters, f_1 and f_2, would cause conflict to filter g_1 or not. The execution of all steps is explained below.

Based on Step 1, all filters are represented as filter spaces. The filter spaces of filters f_1, f_2, and g_1 are the rectangles shown in Fig. 4.

Based on Step 2, the packet space is divided in the first dimension (SrcIP) at all the boundaries of all the filter spaces. The packet space in Fig. 4 is divided into seven sub-spaces, S_0–S_6, in the SrcIP dimension shown in **Fig. 15**.

Based on Step 3, all sub-spaces, S_0–S_6, are divided in the next dimension (DesIP) at all the boundaries of all the filter spaces they are located in. For instance, a part of $\mathscr{S}(f_1)$ is in sub-space S_1; therefore, S_1 is divided at all the boundaries of $\mathscr{S}(f_1)$ in the DesIP dimension. Sub-space S_1 is divided into three sub-spaces, S_{10}–S_{12}, as shown in **Fig. 16**. In addition, sub-space S_3 is divided at all the boundaries of $\mathscr{S}(f_1)$, $\mathscr{S}(f_2)$, and $\mathscr{S}(g_1)$. As a result, sub-space S_3 is divided into seven sub-spaces, S_{30}–S_{36}, as shown in **Fig. 17**.

Based on Step 4, after the packet space in Fig. 4 is divided in all the dimensions, the entire packet space is divided into cells from e_0 to e_{24}, as shown in **Fig. 18**.

```
Input:  Combination C[f1, f2, ... , fK],  Filter g
Output: RFCS,  count
Algorithm BottomUp ( C[f1, f2, ... , fK],  g,  RFCS,  count)
 1:  Initialization  {
 2:      count=0;    RFCS={ Φ };    newc={ Φ };
 3:      for each i∈[1, K] do
 4:        T[i]={ Φ };
 5:  }
 6:  generate K combinations C[f1], C[f2],...,C[fK]
     from C[f1, f2, ... , fK];
 7:  for each combination C[fi] in {C[f1],C[f2],...C[fK] } do  {
 8:      Conflict=DecideConflict(C[fi],  g);
 9:      count ← count+1;
10:      if ( ( Conflict=Redundancy error ) or
             ( Conflict=Shadowing error ) )  then
11:                Add C[fi] to RFCS;
12:      else if( (Conflict=Generalization warning) or
                  (Conflict=Correlation warning) )  then
13:                Add C[fi] to T[1];
14:  }
15:  for each s∈[2,  K] do
16:    for each combination Cj in T[s-1] do
17:      for each filter fi (i∈[1, K]) do  {
18:          newc=Add fi in Cj (all filters in Cj precede
                   filter fi) to generate a new combination;
19:          Conflict=DecideConflict(newc,  g);
20:          count ← count+1;
21:          if ( (Conflict=Redundancy error ) or
                 (Conflict=shadowing error) )  then
22:                Add newc to RFCS;
23:          else if ( (Conflict=Generalization warning) or
                       (Conflict=Correlation warning))  then
24:                Add newc to T[s];
25:      }
26:  Output  RFCS,  count;
End of BottomUp Algorithm
```

Fig. 14 *RFM-B*: Bottom-up algorithm for detecting *RFC*s.

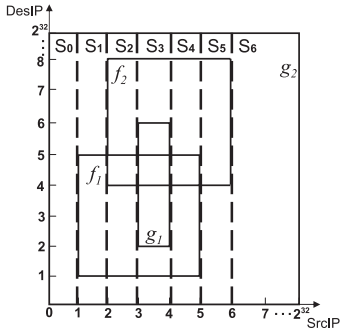


Fig. 15 Division in SrcIP dimension.

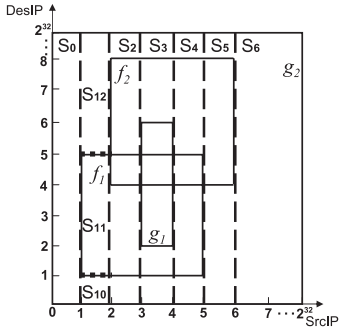


Fig. 16 Division of S_1 in DesIP dimension.

Based on Step 5, each filter space is represented by the following cell sets:

$\mathscr{S}(f_1)$={ $e_2, e_5, e_6, e_{10}, e_{11}, e_{12}, e_{17}, e_{18}$ }

$\mathscr{S}(f_2)$={ $e_6, e_7, e_{12}, e_{13}, e_{14}, e_{18}, e_{19}, e_{22}$ }

$\mathscr{S}(g_1)$={ e_{11}, e_{12}, e_{13} }

$\mathscr{S}(g_2)$={ $e_0, e_1, ..., e_{24}$ }

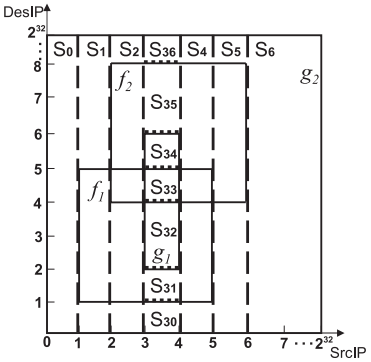


Fig. 17 Division of S_3 in DesIP dimension.

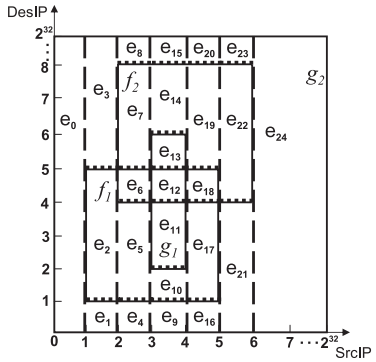


Fig. 18 Division of packet space in all dimensions.

Then $\mathscr{S}$'(C[f_1, f_2]) is represented as follows:

$$\mathscr{S}\text{'(C}[f_1, f_2]) = \mathscr{S}(f_1) \cup \mathscr{S}(f_2)$$
$$= \{e_2, e_5, e_6, e_7, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_{17}, e_{18}, e_{19}, e_{22}\}$$

Based on Step 6, $\mathscr{R}$'(C^K, g_1) is calculated by using the inclusion relationships of the cell sets. The calculation of $\mathscr{R}$'(C^K, g_1) is as follows:

$$\mathscr{S}\text{'(C}[f_1, f_2]) = \mathscr{S}(f_1) \cup \mathscr{S}(f_2)$$
$$= \{e_2, e_5, e_6, e_7, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_{17}, e_{18}, e_{19}, e_{22}\}$$
$$\mathscr{S}(g_1) = \{\ e_{11}, e_{12}, e_{13}\ \}$$

Because $\mathscr{S}$'(C[f_1, f_2]) $\supset \mathscr{S}(g_1)$, the $\mathscr{R}$'(C^K, g_1) is equal to *Inclusion*1.

5.4 SIERRA Tree

We used a SIERRA tree data structure to obtain the cells. A SIERRA tree consists of nodes and leaves, and the structure of a node in the SIERRA tree is very similar to the structure of a node in a multi-way tree [20]. As can be seen in **Fig. 19**, each node contains j values $(b_1, b_2, ..., b_j)$ and $j+1$ pointers $(P_0, P_1, ..., P_j)$. j values specify all the boundaries of all the filter spaces in one dimension, and $j+1$ pointers represent $j+1$ sub-spaces after the entire packet space has been divided at j boundaries. For example, since the boundaries of all the filter spaces in the SrcIP dimension in Fig. 4 are {1, 2, 3, 4, 5, 6}, the corresponding node to the SrcIP dimension is represented as shown in **Fig. 20**.

"Parent" is the pointer that connects the parent node to the current node.

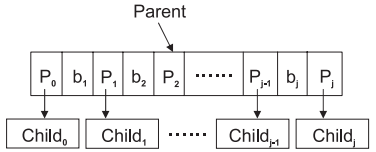


Fig. 19 Node structure.

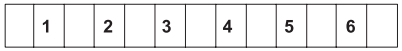


Fig. 20 Corresponding node.

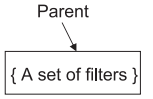


Fig. 21 Leaf structure.

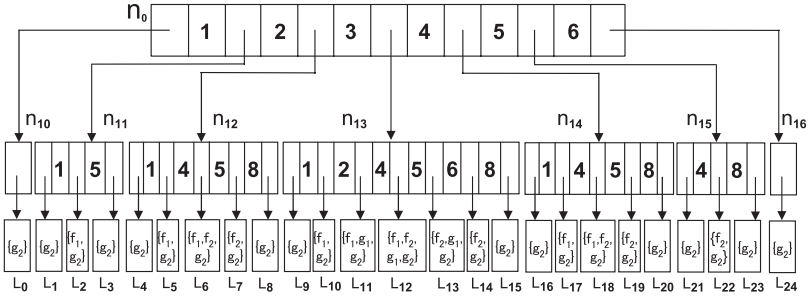


Fig. 22 SIERRA tree.

"$Child_i$" $(i \in [0, j])$ is a child of the current node.

The structure of a leaf is very simple (**Fig. 21**) and each leaf represents a cell. The set of filters in a leaf represents the corresponding cell to which the filters belong. The SIERRA tree of the firewall policy in Fig. 1 is shown in **Fig. 22**.

In the SIERRA tree in Fig. 22, root node n_0 represents the division of the entire packet space in the first dimension (SrcIP) shown in Fig. 15. The entire packet space in Fig. 15 is divided into seven sub-spaces, S_0–S_6, and these correspond to seven child nodes, n_{10}–n_{16}, of root node n_0 in the SIERRA tree.

Nodes n_{10}–n_{16} represent the division of sub-spaces S_0–S_6 shown in Fig. 15 in the second dimension (DesIP). For example, node n_{11} represents the division of sub-space S_1 in the DesIP dimension. Because sub-space S_1 is divided into three sub-spaces, S_{10}–S_{12}, at the boundaries of {1, 5} in the DesIP dimension in Fig. 16, the values of node n_{11} are {1, 5}, and node n_{11} has three leaves L_1–L_3.

In this example, the DesIP dimension is the last dimension, and after sub-spaces S_0–S_6 are divided, the sub-spaces obtained from the division in the DesIP

dimension are leaves. Leaves L_0–L_{24} of the SIERRA tree correspond to cells e_0–e_{24} in Fig. 18.

Since every leaf in the SIERRA tree represents a cell, the filter space represented as the cell set can be represented alternatively by a leaf set. The corresponding leaf sets of $\mathscr{S}(f_1)$, $\mathscr{S}(f_2)$, $\mathscr{S}(g_1)$, and $\mathscr{S}(g_2)$ are as follows:

$\mathscr{S}(f_1)$={ $L_2, L_5, L_6, L_{10}, L_{11}, L_{12}, L_{17}, L_{18}$ }

$\mathscr{S}(f_2)$={ $L_6, L_7, L_{12}, L_{13}, L_{14}, L_{18}, L_{19}, L_{22}$ }

$\mathscr{S}(g_1)$={ L_{11}, L_{12}, L_{13} }

$\mathscr{S}(g_2)$={ $L_0, L_1, ..., L_{24}$ }

In addition, $\mathscr{R}'(C^K, g_1)$ can be calculated by using the leaf sets the same as the cell sets.

6. Experiments and Considerations

6.1 Prototype System

We implemented the proposed methods in two software prototype systems, A and B. The two prototype systems were implemented on an Intel Celeron (R) 3.06-GHz CPU with 512 MBytes of RAM. Each system contained two subsystems.

Prototype system A contained a generating SIERRA tree sub-system and an *RFM-T* sub-system. The generating SIERRA tree sub-system generated the SIERRA tree for all the given filters and represented each filter by using a set of leaves. The *RFM-T* sub-system generated all the combinations that should be assessed based on the top-down algorithm, and then determined which combination could cause conflict and whether any combination was an *RFC*.

Prototype system B contained a generating SIERRA tree sub-system and an *RFM-B* sub-system. The generating SIERRA tree sub-system that was the same as prototype system A's. The *RFM-B* sub-system generated all the combinations that should be assessed based on the bottom-up algorithm, and then determined which combination could cause conflict and whether any combination was an *RFC*.

6.2 Experiments and Considerations

We used three combinations of filters, C1–C3, and a filter, g, as the input for the two prototype systems, and did two experiments to evaluate the two *RFC* finding methods (*RFM-T* and *RFM-B*) and the conflict detection method (*CDM*).

C1: Combination C1 consisted of K filters, and each individual filter, f_i ($i \in [1, K]$), in C1 caused a shadowing or a redundancy error to filter g. This combination of filters represents the worst case of *RFM-T* and the best case of *RFM-B*.

C2: Combination C2 consisted of K filters, and each individual filter, f_i ($i \in [1, K]$), in C2 caused a generalization or a correlation warning to filter g, and no other combinations produced by filters f_1–f_K caused any shadowing or redundancy error to filter g in addition to C2. This combination of filters represents the worst case of *RFM-B* and the best case of *RFM-T*.

C3: Combination C3 consisted of K filters, each filter caused an error or a warning or did not cause any conflict to filter g. The number of each kind of filters is about a third of K. This combination of filters represents a practical firewall policy.

Experiment 1: We wanted to compare the effectiveness of the two *RFC* finding methods, i.e., *RFM-T* and *RFM-B* in this experiment. *RFM-T* and *RFM-B* are algorithms that generate many combinations to find all *RFC*s to filter g from a given K-filters combination. Hence, we think the effectiveness of *RFM-T* and *RFM-B* is determined by the number of combinations that are generated by them. Therefore, we executed prototype system A and B by using combinations C1–C3 and filter g as input, and measured the number of combinations generated by the *RFM-T* sub-system of prototype system A, and the number of combinations generated by the *RFM-B* sub-system of prototype system B. The results for both are shown in **Fig. 23**.

Considerations from Experiment 1: From the results obtained from experiment 1, we found that if the combination C2 was used as the input for prototype system A, the number of combinations generated by the *RFM-T* sub-system was directly proportionate to the number of Ks. However, if the combination C1 was used as the input for prototype system A, the number of combinations shown in Fig. 23 increased very rapidly because the *RFM-T* generated duplicate combinations.

If the combination C2 was the input for prototype system B, the number of combinations did not increase as rapidly because *RFM-B* did not generate du-

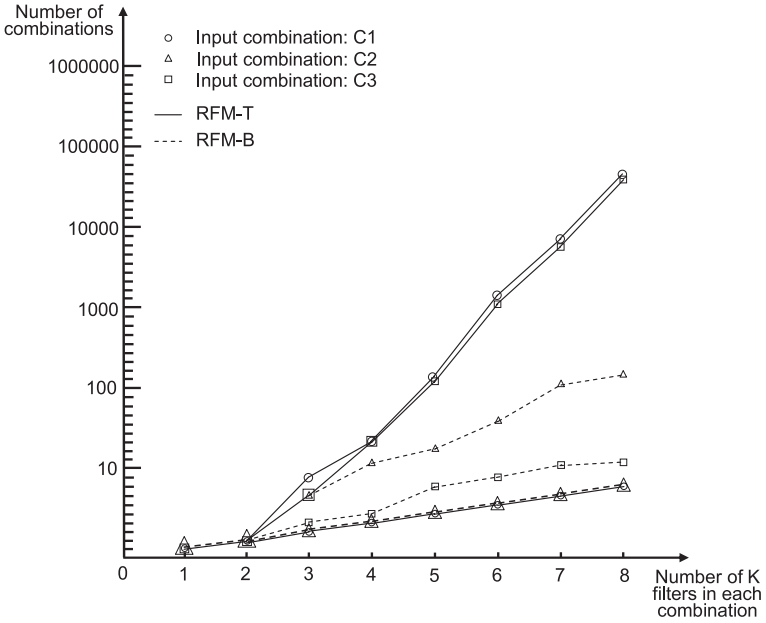


Fig. 23 Number of combinations produced by both prototype systems.

plicate combinations. Further, if the combination C1 was the input for prototype system B, the number of combinations generated by the *RFM-B* sub-system was also directly proportionate to the number of *K*s.

When both the input combinations were the best case for the *RFM-T* and *RFM-B* sub-systems, the two sub-systems generated the same number of combinations. When both the input combinations were the worst case for the *RFM-T* and *RFM-B* sub-systems, the *RFM-T* sub-system generated combinations that included many duplicate combinations, while the *RFM-B* sub-system generated combinations that did not include duplicate combinations. Therefore, we think the *RFC* finding method of *RFM-B* is better than that of *RFM-T*.

Experiment 2: We wanted to evaluate whether *CDM* could be used in a practical firewall policy in this experiment. Because we found that *RFM-B* outperformed *RFM-T* according to the results from experiment 1, we executed prototype system B by using the combination C3 and filter *g* as the input to

The number of filters in combination C3	10	15	20	25	30	35	40
Time (Second)	0.016	0.022	0.027	0.04	0.081	0.2	0.271
Memory (KB)	94.145	250.242	393.153	459.878	527.485	584.437	697.922
The number of combinations	11	16	21	37	85	230	318
Average time	0.0014	0.0014	0.0013	0.0011	0.0009	0.0009	0.0009

Fig. 24 CPU time, Memory usage, number of combinations, and average CPU time of prototype system B.

evaluate whether *CDM* could be used in a practical firewall policy. Because the following three items would be changed when the number of filters in C3 was changed, we measured them and the results are given from the second line to the fourth line in **Fig. 24**. The last line in Fig. 24 means the average CPU time needed to determine whether a combination causes conflict and whether it is an *RFC*.

Item 1: CPU time to execution prototype system B.

Item 2: Memory usage to make the SIERRA tree by generating SIERRA tree sub-system.

Item 3: Number of combinations produced by *RFM-B* sub-system.

Considerations from Experiment 2: From the last line in Fig. 24, we found that the calculation based on *CDM* to determine whether a combination causes conflict and whether a combination is an *RFC* took 0.0014 seconds at most when the number of filters in a combination ranged from 10 to 40. We think that these results are acceptable. Since a practical firewall policy contains many disjoint filters, the results for CPU time, memory usage, and the number of combinations generated by *RFM-B* will be better than those in Fig. 24; therefore, we think *CDM* and *RFM-B* can be applied in practical firewall policies.

7. Conclusion

We developed a conflict detection method (*CDM*) and two *RFC* finding methods (*RFM-T* and *RFM-B*) to determine whether conflicts are caused by a combination of filters or not and to find all *RFC*s from a given combination of filters. The experimental results indicated that *CDM* and *RFM-B* could be applied to practical firewall policies to detect all *RFC*s. Our future research plans include

optimizing the top-down and bottom-up algorithms, and then extending the proposed methods to detect conflicts caused by combinations of filters in distributed firewall policies.

Acknowledgments This research was partially supported by the Ministry of Education, Culture, Sports, Science and Technology, Scientific Research (C) 18500050 and by the Telecommunications Advancement Foundation.

References

1) Wool, A.: A Quantitative Study of Firewall Configuration Errors, *Computer*, Vol.37, No.6, pp.62–67 (June 2004).
2) Al-Share, E. and Hamed, H.: Modeling and Management of Firewall Policies, *IEEE eTransactions on Network and Service Management*, Vol.1-1 (Apr. 2004). http://www.etnsm.org/
3) Al-Share, E., Hamed, H., Boutaba, R. and Hasan, M.: Conflict Classification and Analysis of Distributed Firewall Policies, *IEEE Journal on Selected Areas in Communication*, Vol.23, No.10, pp.2069–2084 (2005).
4) Yuan, L., Mai, J., Su, Z., Chen, H., Chuah, C.N. and Mohapatra, P.: FIREMAN: A Toolkit for Firewall Modeling and Analysis, *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA (May 2006).
5) Baboescu, F. and Varghese, G.: Fast and scalable conflict detection for packet classifiers, *Computer Networks*, Vol.42, No.6 (2003).
6) Hari, A., Suri, S. and Parulkar, G.: Detecting and resolving packet filter conflicts, *Proceedings of IEEE INFOCOM 2000*, pp.1203–1212, Tel Aviv, Israel (Mar. 2000).
7) Liu, A.X. and Gouda, M.G.: Complete Redundancy Detection in Firewalls, *Proc. 19th Annual IFIP Conference on Data and Applications Security*, pp.196–209 (2005).
8) Matsuda, K.: A Packet Filtering Rules Compression by Decomposing into Matrixes, *IPSJ Journal*, Vol.48, No.10, pp.3357–3364 (2007).
9) Hazelhurst, S., Fatti, A. and Henwood, A.: Binary decision diagram representations of firewall and router access lists, Technical Report TR-Wits-CS-1998-3, Department of Computer Science, University of Witwatersrad, Sth. Africa (Oct. 1998).
10) Hazelhurst, S., Fatti, A. and Henwood, A.: Algorithm for improving the dependability of firewall and filter rule lists, *DSN'00: Proc. 2000 International Conference on Dependable Systems and Networks* (2000).
11) Mayer, A., Wool, A. and Ziskind, E.: Fang: A Firewall Analysis Engine, *Proc. 2000 IEEE Symposium on Security and Privacy* (May 2000).
12) Mayer, A., Wool, A. and Ziskind, E.: Offline firewall analysis, *International Journal on Information Security*, Vol.5, No.3, pp.125–144 (2006).
13) Wool, A.: Architecting the Lumeta Firewall Analyzer, *Proc. 10th USENIX Security Symposium* (Aug. 2001).
14) Eronen, P. and Zitting, J.: An Expert System for Analyzing Firewall Rules, *Proc. 6th Nordic Workshop on Secure IT-Systems* (*NordSec 2001*) (Nov. 2001).
15) Yin, Y., Bhuvaneswaran, R.S., Katayama, Y. and Takahashi, N.: Implementation of Packet Filter Configurations anomaly Detection System with SIERRA, *Proc. 7th International Conference on Information and communications Security* (*ICICS2005*), LNCS Vol.3783, pp.467–480 (Dec. 2005).
16) Yin, Y., Bhuvaneswaran, R.S., Katayama, Y. and Takahashi, N.: Inferring the Impact of Firewall Policy Changes by Analyzing Spatial Relations between Packet Filters, *Proc. 2006 IEEE Int. Conf. on Communication Technology* (*ICCT2006*), ISBN: 1-4244-0800-8 Vol.I, pp.203–208, Nov. 27–30 (2006).
17) Yin, Y., Bhuvaneswaran, R.S., Katayama, Y. and Takahashi, N.: Analysis Methods of Firewall Policies by Using Spatial Relationships Between Filters, *2007 IEEE International Conference on Signal Processing, Communications and Networking* (*ICSCN 2007*). pp.348–354, Feb 22–24 (2007).
18) Takahashi, N.: A Systolic Sieve Array for Real-time Packet Classification, *IPSJ Journal*, Vol.42, No.2, pp.146–166 (2001).
19) Comer, D.S.E.: *Internetworking With TCP/IP*, Vol.I, Principles, Protocols, and Architecture 5th Ed.
20) Onald, D. and Knuth, E.: *Art of Computer Programming*, Vol.3: Sorting and Searching (2nd Ed.), Addison-Wesley Professional, Boston, MA (1998).
21) Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C.: *Introduction to Algorithm*, Second Edition, The MIT Press (Sep. 1, 2001).
22) The FreeBSD Documentation Project. Ipfw, http://www.freebsd.org/doc/en-US.ISO8859-1/books/handbook/firewalls-ipfw.html
23) PF: Packet Filtering. http://www.openbsd.org/faq/pf/filter.html
24) Andereasson, O.: Iptables Tutorial. http://iptables-tutorial.frozentux.net/iptables-tutorial.html

(Received November 30, 2007)
(Accepted June 3, 2008)
(Released September 10, 2008)

Yi Yin was born in 1978. She received her B.E. in Computer Information System Technique in 2001 from Southeast University, China, and the M.E. in Computer Science and Engineering in 2005 from Nagoya Institute of Technology, Japan. She is currently pursuing her Ph.D. in the Department of Computer Science and Engineering of Nagoya Institute of Technology, Japan. Her research interests include firewall anomaly detection and network security. She is a student member of IPSJ and IEICE.

Yoshiaki Katayama received his B.E., M.E, and D.E in computer science from Osaka University. He worked at Information Technology Center, Nara Institute of Science and Technology (NAIST) from 1994 to 2003. He is now an associate professor of Graduate School of Engineering, Nagoya Institute of Technology. His research interests include distributed algorithms, network applications and ubiquitous computing. He is a member of IPSJ, IEICE, ACM, and IEEE Computer Society.

Naohisa Takahashi is a Professor of the Department of Computer Science at Nagoya Institute of Technology, a position he has held since 2001. Prior to coming to NIT, he was engaged in research on parallel processing, software engineering and network computing at NTT Laboratories for 25 years. He received B.E. and M.E. degrees in electrical engineering from the University of Electro-Communications, Tokyo, Japan, in 1974 and 1976, respectively. He received a doctorate in computer science in 1987 from Tokyo Institute of Technology. His recent research interests are network computing, ubiquitous computing, geographical information systems and e-learning systems. Dr. Takahashi is a member of the IEEE, the Association for Computing Machinery, the Information Processing Society of Japan, the Japan Society for Software Science and Technology, and the Database Society in Japan.