UNIVERSITÉ CATHOLIQUE DE LOUVAIN
FACULTÉ DES SCIENCES APPLIQUÉES
DÉPARTEMENT D'INGÉNIERIE INFORMATIQUE

# Assessment software development for distributed firewalls

Promoteur :
Prof. Olivier BONAVENTURE

Mémoire présenté en vue de l'obtention du grade d'**Ingénieur Civil** en **Informatique** par Damien LEROY

Louvain-la-Neuve
Année académique 2005–2006

# Acknowledgements

**Prof. Olivier Bonaventure, my supervisor**
for suggesting such an interesting research topic,
for being a very helpful supervisor with lots of good advice, ideas and supports all through the project,
for the main proofreading.

**M. Freddy Gridelet (LS/SGSI/SISY) and M. Pierre Reinbold (FSA/INGI),**
for supplying some real firewall configurations.

**Gregory Culpin and Sébastien Barré, some friends**
for more proofreading.

# Contents

# List of Figures

# List of Tables

# Introduction

Network security, a subject that was barely brought up ten years ago, is these days one of the hottest topics of many technology information publications. This evolution can be viewed as a direct consequence of the great expense of the Internet in our everyday life. The Internet is a marvellous technological advance that allows to buy a book, to phone your friends, to obtain the last news or to read the diary of someone at the other side of the world in a few seconds. But Internet is also a major danger where thousands of people have the ability to destroy information and threaten your private life.

Firewalls are pieces of the Internet security jigsaw. They provide protection to network devices by filtering what is entering and leaving. The protected devices are generally networks of several computers, but firewalls can also be implemented in end-computers, inside the application layer or below. In this work, we will especially deal with firewalls that work inside the network and transport layers.

When several neighbouring firewalls share a part of their policy, it is said that the firewalls are part of distributed firewalls. These shared policies are the subject of this thesis. More precisely, the thesis is about the analysis and the evaluation of these policies.

## Objectives of the work

First, let us start with the global objective of the work. This is to conceive a software application that takes as input two firewall configurations and that outputs a complete overview of the policies implemented between these firewalls. The tool has also to provide some hints to detect potential anomalies in these policies.

The configurations considered by the analyser have to be expressed in a common firewall configuration language that can represent nearly all the possible network configurations. Our language is XML-based and has been built up from scratch.

Another specification is that the application has to be able to import any configuration file of, at least, one of the best-known firewall configuration languages. The supported language is *Netfilter/iptables*, the most used firewall implementation in Linux systems. So the *iptables* configuration files must be parsed into the XML language for firewalls that have just been mentioned.

When the application that explores the policies has been implemented, the ultimate test is to import configurations of large network firewalls and to translate them into our language. Once

it is done, these configurations have to be analysed by our application to discover the policies. Then, our approach has been to allow to check these policies by hand or automatically. All the policies of the distributed firewalls can be explored by hand using a policy tree. A policy tree is a structured tree that provides a quick overview of the policies applied to a specific flow between two hosts. The tool also provides an anomaly detection system that tries to point out the more common anomalies.

A simple graphic user interface (GUI) has been implemented to make the preceding manipulations easier. Among others, it allows to easily explore the policy tree. This GUI will not be discussed in this report but a short manual can be found in Appendix B.

## Plan of the report

This report is divided in five chapters. The first one is an overview of the general problematic. This chapter introduces in more details the distributed firewall issues and defines some concepts useful for the rest of the report. Then, a description of the objectives and hypotheses of this thesis are outlined. Finally it makes a quick review of the publications on firewall analyses.

The second chapter fully describes the intermediate language for firewalls that has been elaborated Some important choices made for its conception are also discussed.

The third chapter explains how the parser has been conceived. The aim of this parser is to translate a firewall configuration from a well-known firewall language to our language. Its global implementation and the *iptables*-specific part of the implementation are explained. The limitations of this parser for *iptables* and other future languages are also discussed.

Chapter four is about the firewall configuration analyser. The chapter has two main parts: First it describes how to build and use the policy tree used for manual anomaly detection. Then, some common anomalies are described and how they are detected in the policy tree is explained.

The last chapter is about the validation of the application. The entire software is tested on real world *iptables* configurations. The anomaly detection mechanisms are evaluated and the way to use the policy tree to detect new anomalies is briefly explained.

Note that French is my native language. Writing this report in English was for me a great challenge. So I apologize for the mistakes that may appear in this text.

Chapter 1

# Distributed firewalls: overview and problem statement

This first chapter begins with a description of what has to be known about firewalls and distributed firewalls to tackle this report. Then the objectives of this work are described in more details as well as the way to fulfil them. Finally, an overview is provided of previous work related to firewall analysis.

## 1.1 Firewall overview

### 1.1.1 Simple firewalls

A firewall can be viewed as a box at the boundary of a network which is traversed by all the packets from the outside to the inside of the network and from the inside to the outside. This box has the ability to check the content of the data flow and to determine if it has to be accepted or rejected. Firewalls can be used to protect any network device, from a single personal computer to networks containing thousands of computers or more. In the case of a personal computer firewall, the filtering may be performed by application level software. On the other hand, in medium and large networks, firewall systems are often included in border routers.

A firewall can perform filtering on lots of network properties. These properties usually come from the network headers of packets. The IPv4 header offers interesting fields for filtering like protocol or source and destination addresses. Other fields regularly used are the source and destination ports of the TCP and UDP headers. However nearly all the possible fields or information inside packets can be interpreted. Filtering can be based on the input interface or MAC address and even on the operating system that has sent the packet[Palm 04].

Anyway firewalls do not just make decisions of accepting or dropping packets. Firewalls may also be in charge of network address translation or application-layer services such as proxies or mail filters. Packet modifications for QoS or VLAN may also be performed by firewall softwares. These background roles are numerous and differ from an implementation to another. The more important ones will be discussed in the next sections.

A typical network of medium size is represented at Figure 1.1. The letters between brackets in

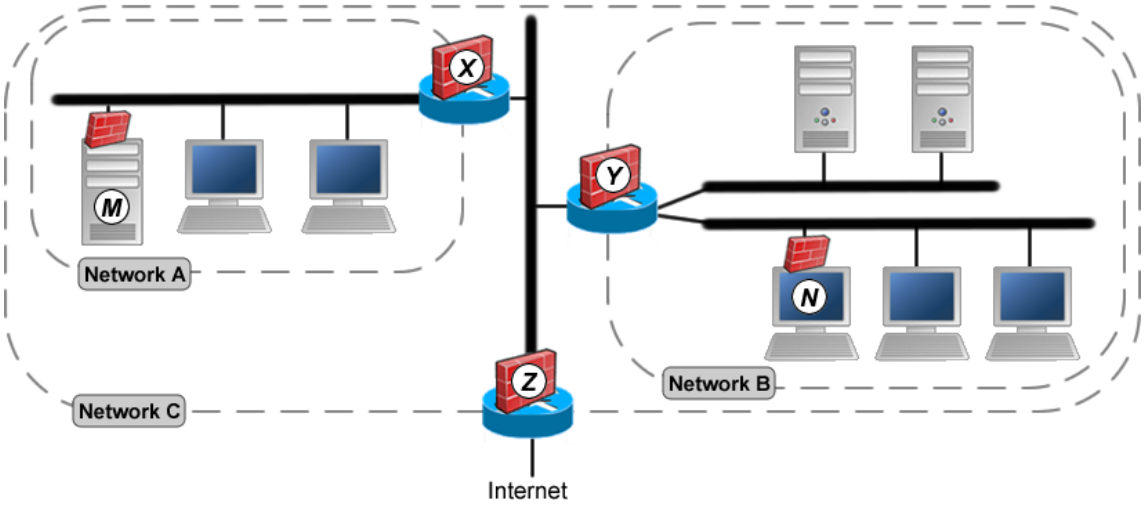



**Figure 1.1 –** Different types of firewalls in a typical network

this section refer to it. Firewalls are represented by brick walls. They can usually be classified in three main categories depending on their location in the network. First, firewalls may be located at the border, as a gateway (Z) between the LAN (C) and the outside. They are also used to protect a subnetwork from another, between two departments in a company for instance. These firewalls are often implemented inside gateway devices (X, Y) between the two subnets (A, B). Finally, they can be implemented on servers (M) or personal computers (N) to protect efficiently themselves.

These firewalls are not redundant. All of them may have a specific role in the security of the network. This kind of structure is essentially useful in networks where computers cannot trust each other. It is nearly the case in every network: each computer may be a potential source of intrusion. The number of firewalls and their positions depend on a lot of parameters and need a detailed investigation.

As a last consideration it can be assumed that a firewall has always two or more interfaces. These interfaces link the firewall with two categories of networks that can be called inside and outside networks. Inside networks are the networks that the firewall is supposed to guard, it generally belongs to them. Outside networks are the networks firewall is protecting from. They generally lead to the network default route (0.0.0.0/0). There is usually only one outside network associated with a firewall and one or more inside networks. Note that in the case of an end-computer firewall, the inside network is the host itself.

### 1.1.2 Distributed firewalls

Distributed firewalls are a set of firewalls that are managed together with the objective to consolidate their security policy. This concept is rather abstract since the relationship between two firewalls may be restrained to an agreement about the policy to apply. In the network represented at Figure 1.1, if each firewall is administrated independently of the others, incoherences may appear. The aim of distributed firewalls is to avoid those incohences as much as possible.

Some distributed firewall systems are centralized. It means that a central management node deals with all the firewalls of the network [Bell 99]. The advantages are that only one configuration is needed and the overall policy is coherent if well implemented. Actually this kind of scheme still

suffers from several problems and is not yet widely used [Ches 03]. So the distributed firewalls considered in this thesis refer to independent firewalls that are administered one by one with a special attention about coherence; they are also called decentralized distributed firewalls.

### Potential issues

Since each firewall is administrated independently from the other ones, some issues may quickly occur. Here are some examples based on the network illustrated at Figure 1.2.

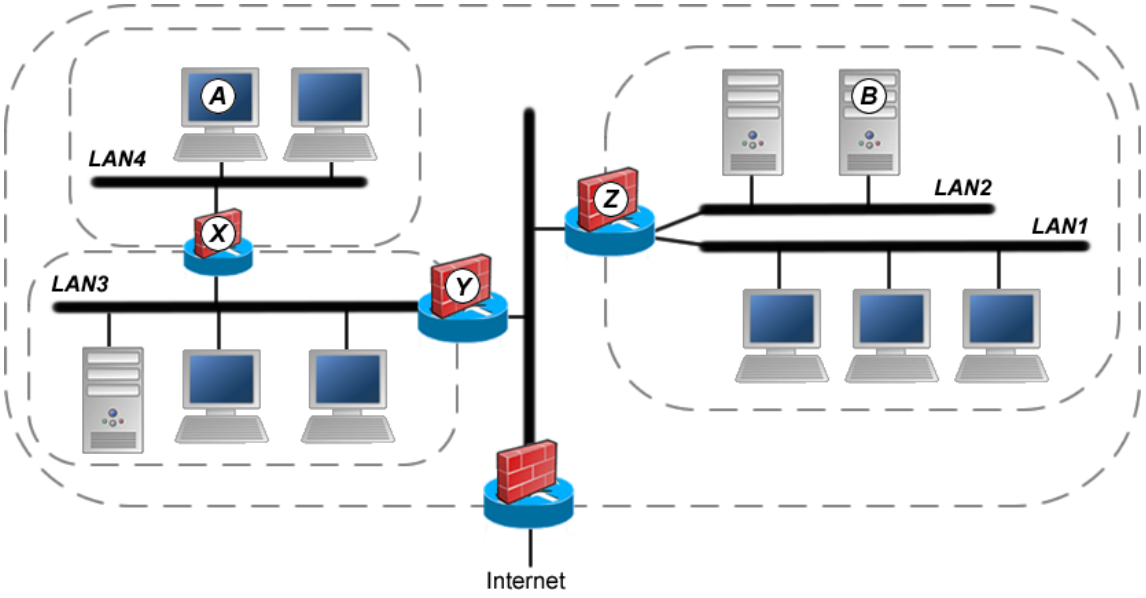



**Figure 1.2 –** A network in which distributed firewalls may generate anomalies

The network administrator of LAN3 wants to prevent the users of its subnet from going on the external website WWW.SITE.COM. A solution could be to block all these requests in firewall Y. However, it is no use for requests from A to the web server to go through LAN3 since they will be dropped in Y. These should be rejected in firewall X at once.

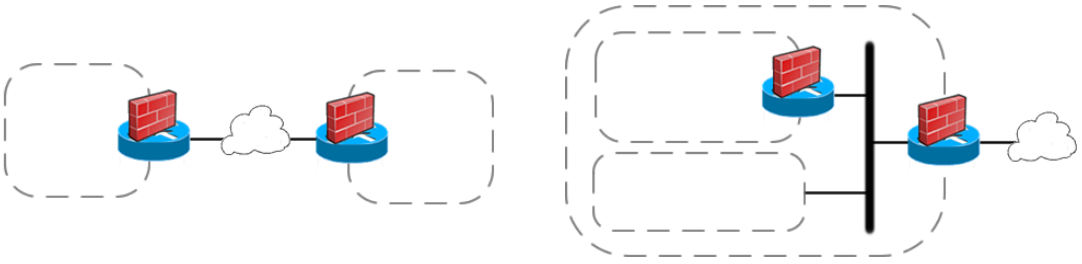
The host B is a mail server that has been built up to answer to user requests of the entire network. So firewall Z has been configured to accept requests from LAN1, LAN3 and LAN4 to B. However LAN3 and LAN4 do not deal with B anymore and the firewall Y rejects the mail requests to this destination. In the current configuration of the network, no real problem has to be noticed. Nevertheless it is not a good idea to let Z accept the requests from LAN3 and LAN4 addresses. Furthermore, an evil-minded person could use it to spoof their addresses and attack the mail server. This example shows that a small change in the policy of a subnet may have an impact on the configuration of a whole distributed firewall system.

### Architecture of distributed firewalls

As seen in the example of the preceding section, the firewall positioning in a distributed firewall system may be rather complex. They can be located on the same link as other firewalls, with their network included in a network protected by another firewall or other configurations. To clear it up, from now on, the analysis of distributed firewalls will always be made by considering two

firewalls. This is not a restriction since for three neighbour firewalls U, V and W, if the pairs of firewalls U-V, V-W and W-U are coherent, it can be assumed that U-V-W is also coherent. Another assumption is that only neighbour firewalls can be directly compared. On Figure 1.2, X and Y can be compared, as well as Y and Z. To compare X and Z, it is necessary to merge one of the two firewall configurations with the Y one.

Finally, the positioning between two neighbour firewalls can be classified in two categories. The first one, represented at Figure 1.3a, is *face-to-face*. With this symmetric layout each firewall can reach the other one through its outside network interface (see 1.1.1). The second category is *one firewall behind the second one* and is represented at Figure 1.3b. In this positioning, one firewall is included in the network protected by the other one. The first firewall can join the second one through its outside network interface. The second one has to use one of its inside network interfaces to join the first firewall. The difference between these two categories will have a significant importance in a later part.



(a) Face-to-face (b) One firewall behind the second one

**Figure 1.3 –** Various positioning of firewalls

### Administering difficulties

The distributed firewalls discovered in the previous examples reveal the complexity that may appear in large networks using firewalls. The administration of distributed firewalls in a large network is especially difficult for the following reasons:

- The policy implemented in a firewall may be difficult to read and interpret correctly by a human. It is especially the case in large networks that use complex firewall configurations and where the number of rules can exceed one thousand.
- When the policy implemented in a single firewall is still clear, understanding the whole policy of a distributed firewall system is usually more complicated. Among other issues, the network architecture and the behaviour of each network have to be well understood to build up strong systems.
- In large networks or in highly fragmented ones, several administrators are generally in charge of the network security. Each of them is usually responsible of one or several subnets. An acceptable overall configuration can only be realized if there is a good communication between the persons in charge of the firewalls.
- In a single network, firewalls can be implemented in various hosts and operating systems. To understand these firewall configurations, people need to have good knowledge on them. This does not allow a quick overview on a diversified network.

### 1.1.3 Packet filtering implementations

As written previously, firewalls can appear at lots of locations in networks. They can be set up on routers or general-purpose computers. The implementations mainly depend on the manufacturer for routers and on the operating systems for computers. Here is an overview of the best-known implementations of the moment. (inspired from [Zwic 00] and other documentations)

#### *ipchains* and *netfilter/iptables*

*ipchains* and *iptables* are different versions of the same packet filtering system included in Linux kernel since version 2.1. *ipchain* has replaced the old firewall code (*ipfwadm*) used in kernel 2.0 and has been superseded by *iptables* since version 2.4.

*ipchains* has been one of the first filtering system to provide masquerading on Linux. Actually, masquerading is a dynamic network address translation (NAT) system. More precisely, it is the most common type of NAT. It allows someone that has only one public and dynamic IP address to connect more than one computer to a public network. It is also called SNAT for *source NAT* since it translates the source address of the first packet and the next outgoing ones [Netfilt].

Another important concept introduced by *ipchains* is the rule chains. Each rule can be split into three parts: the first one is the chain concerned by this rule, the second one is a set of conditions and the last one is a target chain or action (accept, deny, ...). The rules in a chain are read in order; if an action is taken about a packet, no more rule matches the packet anymore. If a packet belongs to a chain and matches the conditions, its new chain is the target chain or if an action is specified the action is taken. So a packet may pass through several rules before an action occurs. More details about *ipchains* and *iptables* chains will be given in the section 2.4.

Another strength of ipchains and its successor is to be easily extensible. Many extensions are included in the base *iptables* package and are loaded as kernel modules. They provide a lot of new matches and new targets that guarantee more accurate filtering.

*iptables* (or *Netfilter*) is the successor of *ipchains* and has kept its main properties, it is developped inside the *Netfilter/iptables Project* [Netfilt].

#### *ipfilter* (ipf)

*ipfilter* (also known as *ipf*) is another packet filtering and NAT system for Unix. It comes as a part of FreeBSD, NetBSD and some versions of Solaris. OpenBSD does not use it anymore due to licensing problems. *ipf* has also been ported to other Unix operating systems such as SunOS, IRIX or Linux.

*ipfilter* checks its rules in sequence but unlike *netfilter* the last rule matched by a packet is used to determine the fate of it. This behaviour can be avoided by using the *quick* keyword in a rule. It allows a packet that matches the rule to be immediately handled. Rules may be arranged into groups to build up more complex configurations. Groups are defined by a head rule that determines if the rules inside the group have to be checked or not.

The address translation functionality of *ipf* is minimal and weaker than in *netfilter*. Another weakness is that *ipfilter* is also more difficult to extend than *netfilter*.

#### PF packet filter

*PF* packet filter is the firewall used by OpenBSD since version 3.0. It replaces *ipf* that is no more used for licensing issues.

*PF* has kept the main characteristics of *ipf* but has improved its weak points. It has been merged with ALTQ traffic-shaping framework for QoS support. Network Address Translation (NAT) and advanced features have also been integrated into *PF* to allow greater flexibility. The offered functionalities now permit to build feature-rich firewalling devices. So *PF* has already been ported to FreeBSD and NetBSD. [Palm 04]

#### Cisco Access Lists

The most encountered routers in networks are the *Cisco Systems*' one. One of the operating system run by these routers called *Internetwork Operating System* (IOS) uses the *access list* concept for packet filtering.

An *access list* is a serie of rules that instructs the router on what to do with an incoming or outgoing packet. Each rule of an *access list* has three main parts: a number that identifies the list, a *deny* or *permit* decision and a list of conditions to match. A list number is associated with an interface and a direction (in or out). When a packet arrives on an interface and with a direction that corresponds to an *access list* number, the rules identified by this number are processed sequentially. If the packet matches the conditions defined by a rule, the decision (permit or deny) is immediately made. [Seda 01]

Cisco routers obviously implement address translation but NATing is not performed in *access lists*. More information about the NAT capabilities of *Cisco IOS* will not be given in this document. However, a full description of *Cisco IOS* can be found in [Bone 02].

## 1.2 Thesis specifications

### 1.2.1 Global objectives

The preceding section has shown that administrating distributed firewalls is not an elementary operation. The aim of this thesis is to make this task easier and more efficient. The final objective of this work is to design, implement and evaluate an assessment software for distributed firewalls. This software should take as input the configurations of distributed firewalls in order to output a representation of the policy and point out some anomalies.

The secondary objectives are the following:

- The tool has to import distributed firewall configurations. In order to do that, it has to be able to parse the configuration files of at least one well-known filtering system.
- The parser part of the software has to be easily extensible to support new input languages. For this reason, it has to be as much as possible independent from the evaluation part.
- The final application has to be evaluated on a large network configuration.

### 1.2.2 Hypothesis

In order to restrict the problem, some small hypothesis have been made.

The first one is that the network routes to the outside of amm the networks are thoroughly symmetric and unique. In a more formal way, it means that for a host A in the network and a host B outside the network, a packet from A to B always passes through the same gateway firewall to leave the network. Moreover, this firewall is also traversed by the packets from B to A. So, the network architecture represented at Figure 1.4 is not valid. Note that this kind of network structure is almost never encountered since stateful packet filtering is impossible. One solution could be to translate source addresses to the gateways' when packets get out. This problem is out of this work's scope. The entire statement of this kind of configuration is explained in the section 9.4.2 of [Ches 03].

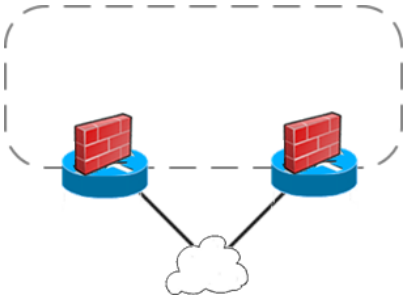


**Figure 1.4 –** A network with two ways out to the same external network

The second hypothesis may appear a bit strange but is important for later suppositions. It assumes that firewalls protect themselves against IP address spoofing. Of course, only recognisable spoofing has to be filtered out. It includes the outgoing packets with a source address corresponding to a external address and incoming packets with a source address corresponding to an address of the network.

The last hypothesis is that no NATing is performed in the firewalls considered. The reasons of this will be widely explained later in section 2.2.3.

### 1.2.3 Steps

To achieve the described objectives, an analysis of the different steps is needed. One requirement is that the firewall evaluation has to be as much as possible independent from the input language. It means that the implementation has to be divided in two main parts: the parsing and the evaluation. It could even be better if an independent parser that supports a new firewall configuration language could be used without changing anything to the previous implementations. So the solution is to design an intermediate language in which the parser has to translate the input files. Next, the evaluation part of the implementation just has to import this configuration and analyse it. With this process, two more constraints have to be appended. The first one is that for most of firewall systems, all configurations could be translated into the intermediate language. The second one is that the language should be easy to parse by the second part of the implementation.

So the major steps of this work are the following:

- Design a language to configure filtering rules on firewalls

- Implement a parser to translate real firewall configurations to the language
- Implement an analyser that evaluates the distributed firewall configurations

## 1.3 Global decisions

### 1.3.1 The programming language

The programming language chosen for this implementation is Java. It has been selected for the following reasons:

- This language is well-known by the author. It permits not to waste time with silly implementation issues.
- Java is a good choice for cross-platform compatibility. Since the tool has to be used by system administrators, it is more appropriate to have the software compatible with exotic systems.
- Java has integrated XML support and is a good choice for tree structures that will be often used in this implementation.

The version of Java SDK used is the 1.4's. It is preferred to version 5 since it has been released a few years ago and is still the version in use in a lot of organisations. So the application has to be run with JRE version $\geq 1.4$.

### 1.3.2 The application's name

A name has been given to the software developped: DFA for *Distributed Firewall Analyser*. It is not very important but it permits not to write "this implementation" during all this report when the context relates to the implementation associated with this work.

### 1.3.3 Notations

Throughout this report, short rule listings will be frequently shown as examples. The way to interpret these listings has to be determined to avoid misunderstanding. When a list of rules is displayed, it is assumed that the first rule matched by a packet determines its fate.

A rule has the following intuitive structure:
`[chain :] condition_list => ACCCEPT|DROP|chain.`
The chains are only used to represent the rules into languages that include this concept. *Accept* or *drop* is obviously the action performed on the packet when it matches. Condition list is a series of conditions separated by a "`&`". All the conditions must be true to match a packet. Each of these conditions is written `field_name[!]=field_value` except for fields that do not need a value. The equals (=) symbol is an abuse of notation since `src_ip=1.0.0.0/8` does not mean that the source address has to be equals to 1.0.0.0/8 but rather that it has to be included in 1.0.0.0/8. The ranges are represented by two boundaries separated with a dash (−).

The rule list shown at Figure 1.5 is a valid example of it. The input chain `forwarded` means that the three rules will be traversed by each forwarded packet. A packet begins by checking the

first rule. If its source IP address is included in 1.0.0.0/8 and if its protocol is TCP, the packet is definitely accepted by this filtering table. Otherwise, the second rule is checked. If the protocol is UDP and if the destination UDP port is contained between 20 and 24 (20 and 24 included), the packet is accepted and the third rule is never checked. Finally, if the packet hasn't matched the two first rules, it checks the third one. There is no condition, so the rule matches all the packets. Thus, the packets that do not match one of the two first rules are dropped.

```
forwarded: src_ip=1.0.0.0/8 & prot=tcp => ACCEPT
forwarded: prot=udp & dst_port=20-24 => ACCEPT
forwarded: => DROP
```

**Figure 1.5 –** A valid example of firewall rule list

### 1.3.4 Data manipulations

The parser and the analyser have to manipulate rule data to perform their job. One of the main complicated operations they have to do is the conjunction (AND operation) of two rules. It involves a conjunction of each of the common fields but doing this operation on fields is not easy. The next items represent the values encountered in rules and explain their properties illustrated with the conjunction operation.

#### IP addresses

What is called an IP address here rather corresponds to an IP subnet. It is defined by a "real" IP address, i.e. four positive integer between lower than 256, and a mask length (CIDR). The CIDR is a number lower or equals to 32 that indicates the number of bits of the IP address corresponding to the network. The CIDR number can easily be translated to and from network or wildcard mask. An example of IP address is 1.0.1.0/24 thus.

Such IP addresses have the property that for two IP addresses A and B, the relation between them can only be one of the following. Note that wherever their values, A and B can be viewed as sets of end-host IP addresses.

- A includes B ($A \supset B$)
- A is included in B ($A \subset B$)
- A is equivalent to B ($A = B$)
- A and B are disjoint ($A \cap B = \emptyset$)

Then if A and B have an intersection, either A includes B or B includes A. To determine the intersection between two IP addresses, in order to do the conjunction of two source IP fields for instance, the analysis of whether one address includes the other has just to be made.

Imagine a list of addresses associated with information. For this example, the information will be a colour: (1.0.0.0/8:blue, 2.0.0.0/8:red). If a new address 1.0.4.0/24:yellow has to be added, the property of IP addresses permits to define the new list as (1.0.0.0/8:blue, 1.0.4.0/24:blue/yellow, 2.0.0.0/8:red). To read the information associated with the address 1.0.4.20, the more specific

address of the list has just to be found, 1.0.4.0/24:blue/yellow here. This example shows how this interesting property should be used and will be used in the next chapter implementation.

#### Ranges

Ranges of values are used in two circumstances: the UDP or TCP port values and AH or ESP *Security Parameter Index* (SPI). Unlike the IP addresses, two ranges can have an intersection without having one range entirely included in the other one. For instance, the intersection of 10-30 and 20-40 is 20-30.

If the same exercise is done for ranges than for IP addresses, our list may look like (0-30:blue, 40-50:red). If (20-45:yellow) is added, the list becomes (0-20:blue, 20-30:blue/yellow, 30-40:yellow, 40-45:red/yellow, 45-50:red). So it is obvious that ranges need more manipulations in such operation. However, calculations are easier on ranges (a couple of integers) than on IpAddresses. More specific algorithms will be explained in the next chapters.

#### Values

Values are obviously easier to manage than addresses or ranges. They are particularly encountered in the interface fields or in the protocol ones.

#### No value fields

The main field that has no value is the *established* one. The operation on this category of field is quite obvious too.

## 1.4 Firewall analysis: state of the art

More and more articles about security are released each year, firewall is one the subject covered by those. Although a lot of publications are about firewall analysis [Al S 03, Hame 05, Scha 04, Al S 04b, Bart 99, Maye 00], a tiny amount of them deals with analysis of firewall groups or distributed firewalls. Here is the review of some interesting articles out before the end of 2005.

### 1.4.1 Single firewall analysis

A lot of tools already exist on Unix systems to manage firewalls but most of them have two main issues: [Scha 04]

- They do not include a functionality to parse any well-known firewall language; they only give a way to export their configuration in these languages. So the firewall configuration has to be built up from scratch into the software. This problem limits their use to simple personal firewalls or small size network ones.
- These tools do not use all the potential of the languages. They limit their use to a closed list of possibilities.

However, two interesting papers on simple firewall analysis have been found. These two works are the ones that have had the most influence on this work.

The first one, [Al S 03] has tried to define with strictness all the possible firewall policy anomalies. To do that, it begins by formally describing a model of rule relations. Next, it lists with the same formalism all the possible firewall anomalies that may occur in a set of rules. The authors describe an anomaly as the existence of filtering rules that may match the same packet or the existence of a rule that may never match any packet. Then an algorithm is proposed to discover these anomalies and to correct them. The authors have implemented an application that is able to detect the anomalies, it is called *Firewall Policy Advisor* (FPA). This article provides a very good analysis of all the simple anomalies that may occur but has two problems. First, FPA does not parse well-known language rule lists. The analysis is made from a very simple meta-language made of five fields : protocol, source IP, destination IP, source port and destination port. A TCP flow is represented by one rule in the initialisation direction. For example, this rule of their language "`tcp 1.1.1.1 0 2.2.2.2 22 accept`" allows a SSH flow between `1.1.1.1` and `2.2.2.2:22`[1] only if the flow is initiated by `1.1.1.1`. The second weakness of FPA is that it is impossible to process more complex rules like the TCP flags or application level filters.

[Scha 04] is an undergraduate dissertation about a visualisation software development for *iptables* rules. The dissertation describes among other things the parsing of the *iptables* rules into an XML language. The application is writen in *C++* and can be extended with new modules. Unfortunately it was not possible to obtain the implementation of the program but the approach described in the report is interesting for this work.

### 1.4.2 Distributed firewalls analysis

The authors of [Al S 03] have written a second article in the same field. [Al S 04a] is about anomaly discovery in distributed firewalls. It consists of a summary of the previous article with a few new sections about inter-firewall anomaly discovery. As in the first article, this one formally defines what they are considering as a anomaly and the algorithm to discover these. Inter-firewall anomaly discovery functionality has also been added to their FPA tool; nevertheless the implementation of the tool is not freely available. This article provides an interesting foundation for anomaly detections but is maybe a little bit too formal and limited to be applied on real large configurations. The inter-firewall anomalies described in this article will be discussed in more details in the appropriate section.

[1] `x.x.x.x:y` is a simplified notation for "the port `y` on host `x.x.x.x`".

Chapter 2

# Definition of a language for firewalls

As previously explained, an intermediate language for firewalls has to be built up first. This language will be an intermediary between the parser part and analyser part of DFA. It has to be expressive enough to represent most of the possible firewall configurations and easily upgradeable to support new firewall features such as advanced IPv6 rules. This chapter first explains the meta-language chosen for the structure of the language. Next, the characteristics of other firewall languages are examined to decide which ones will be integrated in the language and how this will be done. The features considered include the frequently used fields, the actions that differ from pure filtering, the chains, the interfaces and the data flows.

## 2.1 Language structure

### 2.1.1 Representation of the language

The description of firewall configurations in a generic language has to follow a structure. This structure will be the subject of the next section of this chapter but first the way to organise all this information must be decided. Our objective is to obtain files that can be easily parsed. For this purpose, a markup meta-language is the best choice. The first idea that comes is obviously the most famous one: the *eXtensible Markup Language* (XML). This meta-language, recommended by the W3C [W3C], is a general-purpose markup language for creating special-purpose markup languages, capable of describing many different kinds of data. Here are some more reasons to choose XML as meta-language:

- XML parsers are available for most programming languages. Thus XML files can be written and read easily.
- If the language specification and the markers used are well defined, the files generated can be read on all platforms by anyone.
- It is easy to create or edit XML files by hand via a front-end application or a simple text editor. This is useful for debugging, validating and checking how the application performs the translation.

### Short overview of XML

An XML document is above all a text document with – of course – a particular syntax. Figure 2.1 is an example of a simple XML document that represents a (small) library.

```
<?xml version="1.0"?>
<library>
  <book ISBN="1-56592-871-7">
    <title>Building Internet Firewalls</title>
    <author>Elizabeth D. Zwicky</author>
    <author>Simon Cooper</author>
    <author>D. Brent Chapman</author>
  </book>
  <book ISBN="0-201-63466-X">
    <title>Firewalls and Internet Security</title>
    <author>William R. Cheswick</author>
    <author>Steven M. Bellovin</author>
    <author>Aviel D. Rubin</author>
  </book>
</library>
```

**Figure 2.1 –** An example of XML document representing a library

This document is a well-formed XML document and so can be read by any XML parser. Here are some definitions of the XML vocabulary: [Haro 02a]

- An *element* begins with a start-tag (`<element_name>`) and ends with an end-tag (`</element_name>`). In the example, a library, a book, a title and an author are *elements*.
- Everything between a start– and an end-tag is the *content*.
- If there is no content between the two tags, the *element* can be represented by `<element_name />`.
- The *element* `book` is the *parent* of the *element* `author` and is a *child* of the *element* `library`.
- The *element* `library` is the *root* of the XML document.
- `ISBN` is an *attribute* of `book`.

A redundant question is whether to use a new child element or an attribute to hold information. In the previous example, could the title be set as an attribute of a book? There is no ultimate answer to that question. It is commonly said that an attribute is used for metadata about an element while a child element is used to store information itself. Child elements are also chosen when there are more than one entry of the same type to store. In the example, authors cannot be put as an attribute of a book since it is possible to have more than one author for a book. [Haro 02a]

XML also allows more "complex" optional features. Two of them that may be useful are *namespaces* and *Document Type Definitions* (DTD). An XML namespace is a standard for providing uniquely named elements and attributes in an XML document. DTD, as the name indicates, define a precise structure of the content of an XML document. In the example, `author` can be a child of `book` but not a direct child of `library` or `title`. It is also possible to define in DTD

that a book must have one and only one title. More information about theses features is available on the *W3Consortium* website [W3C] or in books about XML [Haro 02a].

### 2.1.2 Language formalisation

Some formalisation is now necessary to define how this language has to be understood:

- Filter rules will appear in the specific order given by the succession of the XML rule elements. This order is total. So between two rules, there is always one of them positioned before the other one.
- The rules have to be checked in sequence, the first one that matches a packet determines the outcome of it.
- For each packet, there is always at least one rule that matches it. So, no default policy is needed.

### 2.1.3 Document definition

First, it is important to notice that the objective of our language is not to use all the capabilities of XML to obtain a perfect robustness. It is only to obtain well-formed XML documents with valid structures. For this reason, XML namespaces and DTD are not used by our language. The XML files are supposed to be generated by our application and thus, a verification of the structure is not needed. Anyway, if a structural error appears it will be detected by the analyser.

The root of the firewall configuration document is an element called `firewall`. The only attribute of this element is the source language of the firewall. This is not an essential information and it is not used in the future. Children of `firewall` are interface definitions and rule tables. Interface definitions describe for each firewall network interface the subnets attached to it. It is required to analyse authorized packet flows. This is detailed in the section 2.3. Rule tables express all the tables in a firewall that are passed through packets, this is to say the filter table, the NAT table, ... In this work, only the filter tables will be considered. More information about this will be supplied in the section 2.2.

Table 2.1 describes the root element as explained in the previous paragraph in a more formal way. The symbols between brackets represent the cardinality of each content in the elements: $(?)$ = zero or one, $(*)$ = zero or more, $(+)$ = one or more, $(1)$ = exactly one. The child elements will be described in next sections.

| **`firewall`** | |
|---|---|
| *Description* | The root element of the document |
| *Attribute* | `srclang`: the language of the original firewall configuration file |
| *Allowed content* | `interface`(+), `filtertable`(1) |

**Table 2.1 –** Definition of the element `firewall`

## 2.2 Filtering, NAT and other tables

As explained in the first chapter, firewalls do not only perform simple packet filtering. Several other roles are generally associated with firewalls. For this project only pure filtering rules are supported. Thus it is important to extract these rules from the firewall configuration. Here are some of functionalities that can be performed by firewalls and the way to extract data from them.

### 2.2.1 Filtering

Packet filtering includes all the rules that inspect packets to determine whether a packet must be accepted or rejected. The rule is applied to all the packets that pass through the firewall. Here is an example of a rule.

```
src_ip=1.1.1.1 & prot=UDP => DROP
```

If the source IP address of the packet checked is equals to `1.1.1.1` and if the protocol is UDP, then the packet is dropped.

The filtering rules are obviously the most interesting ones to determine which packets will be authorized. So they have to be converted in the intermediate language. All the possible fields of rules will be examined in section 2.5. The XML element corresponding to the filter table is described in table 2.2.

| **`filtertable`** | |
|---:|---|
| *Description* | Table that contains all the filtering rules of the firewall. |
| *Attribute* | none |
| *Allowed content* | `frule`(+) |

**Table 2.2 –** Definition of the element `filtertable`

### 2.2.2 Logging

In the same way as packet filtering matches some properties to determine whether accepting or dropping a packet, the target of a rule can be a log file. It is often use to keep track of packets that are not supposed to be sent in a standard data flow. These log files can be analysed afterwards to detect potential attacks or anomalies. For this work, the logged rules have no interest since they do not cause packets to be dropped or accepted.

### 2.2.3 Network Address Translation (NAT)

Network Address Translation is widely used in routers and gateway hosts, however it is less frequently used in end-computer firewalls. NAT proliferation can be viewed as a consequence of the lack of IPv4 addresses. Address translations can be divided in two main categories: [Andr, Netfilt]

**Destination NAT** is used to redirect packets that arrive at a specific port to another host or/and another port. The job of the NAT in this case is to alter the destination address and port

of packets and to reroute them. Port forwarding, load sharing and transparent proxying are examples of the purpose of destination NAT.

**Source NAT** is the category of NAT that is the most frequently used. One of its objectives is to give the opportunity to hosts of a local network to access the Internet by using a single or a few public IP addresses. This is used by many home users to connect more than one computer to the Internet in spite of the single IP address provided by their ISP. This kind of source NAT translates the source address of packets that are leaving the network to the public address of the gateway. Then, it translates back the destination address of reply packets coming from the Internet. Source NAT is generally performed dynamically but can also be done statically with a provided address matching. Source NAT is sometimes called *masquerading*.

### How to consider NAT rules in DFA

The final objective of this thesis is to design a tool that shows which flows will be accepted between two firewalls. Now, let us see how NAT can be dealt in such analysis. In the following paragraphs, *local address* refers the IP address and port in the local area. On the other hand, *public address* corresponds to the same address as it is known on the Internet. The link between the two addresses is the address translation made in the NAT box. The term *flow* includes all the packets with the same source and destination IP addresses and the same source and destination ports.

Our objective is to follow the same flow through several firewalls. However, the problem is that a reference to the source (or destination) address of one packet changes from a firewall table to another. In the example of Figure 2.2, the rules associated with the packet sent is the first one in the left firewall (if NATing is done after filtering) and the second one in the right one. At first sight, one of these solutions could be considered: for each address translation, convert all the address occurrences of a flow into the local or into the public address. But this is not correct. If it is converted into the local address, a conflict may be noticed on Figure 2.2 in the right firewall. If it is converted into the public address, all the filtering that concerned the local area is no more possible. For instance, if the router of the local network decides to block all the UDP packets coming from 192.168.1.10, the rule on the IP address cannot be replaced by the public address since all the hosts are not concerned by the rule.

Another solution could be to replace all the occurrences of translated addresses by a virtual address like "*local address*:*public address*". However, the port translation must also be taken into account, static and dynamic port attribution must be considered. But what to do if the assigned ports are between 1024 and 4096 and that an encountered rule accepts port between 1024 and 2048 and drops the other ones?

Another problem is that NAT configuration is not always contained in the firewall one. In Linux, NAT is generally performed by *Netfilter* and its configuration can be read in the *Netfilter* output. But it may also be performed by *fast-NAT* as well. *Fast-NAT* is implemented inside the IP routing code of the Linux kernel [Andr]. This solution may be more efficient if there is no state to store.

Because of these various possible configurations and the difficulty to interpret them, NAT is not taken into account in the current version of our language. We assume that no address translation is performed in the configuration file supplied. Otherwise, the concerned rules have to be removed from the configuration file.

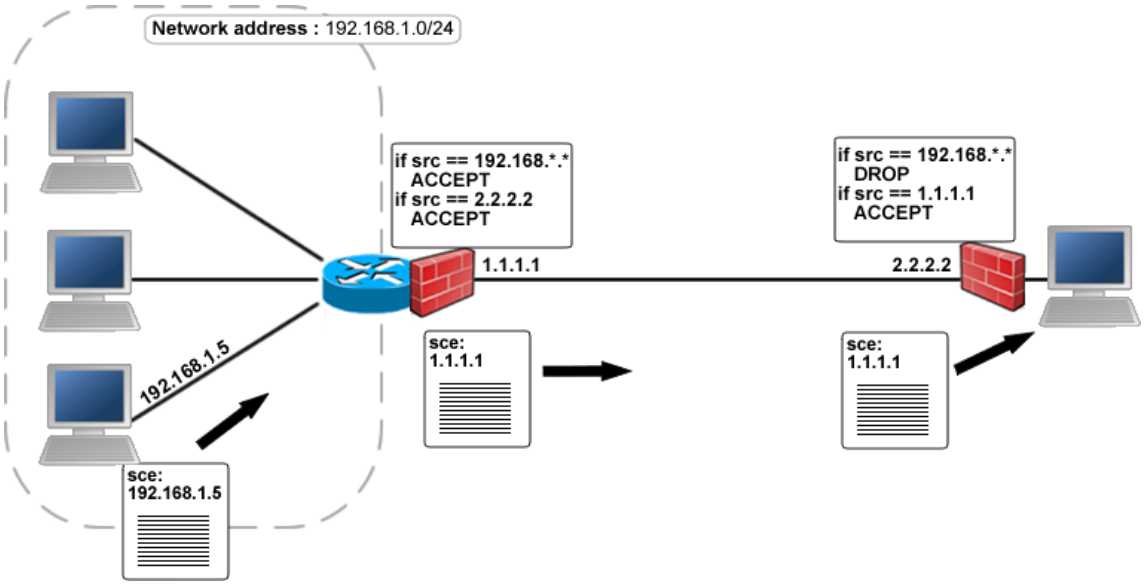



**Figure 2.2 –** Example of a message sent through a NAT

### 2.2.4 Packet alterations

Other encountered rules are those that change packet information. The fields usually modified are the *Type of Service* (TOS), the TCP *Maximum Segment Size* (MSS), the *Time To Live* (TTL) or other significant information for local routing. These rules are not widely used and can actually be ignored since filtering is seldom performed based on this kind of fields. Nevertheless, it could be interesting to control in configurations whether the modified fields are indeed not used in filtering.

## 2.3 Network interfaces and connected subnets

It is important not to forget the Virtual Local Area Networks (VLANs) used in a lot of networks. VLANs are useful to divide networks on a single physical infrastructure. Routing and filtering tables consider these VLANs as separated networks. So a packet transmitted through a VLAN can be considered as transmitted on a specific network.

As explained in the first chapter, simple firewalls usually possess two of more network interfaces; One or more connected to the inside networks and one to the outside network (to 0.0.0.0/0). Exceptions are the firewalls implemented in end-computers. These hosts generally have only one interface.

In most firewall implementations, packets from or to the local host are matched differently. For instance, in *Netfilter* filtering tables, rules that match packets entering in a host begin with `-A INPUT`. Three elementary chains are recognized in filtering tables: `INPUT`, `OUTPUT` and `FORWARD`. Figure 2.3 represents the treatment of these chains. The large ovals symbolize the chain filtering process.

The objective of our language is to be as general as possible. For this reason, the local host is considered as a sub-network behind a virtual interface. Of course, this sub-network contains only a few IP addresses but it is not a problem. Figure 2.4 shows the new version of the filtering

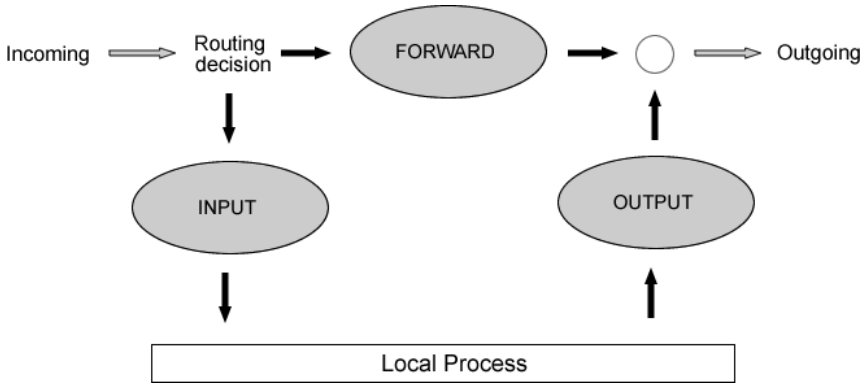



**Figure 2.3 –** Processing of incoming, outgoing and forwarded packets in *Netfilter* filtering tables

process. To distinguish the packets going to the local host from the other ones, these packets are considered as going to the interface `firewall_host`. In the same way, packets from the host are coming from this interface.

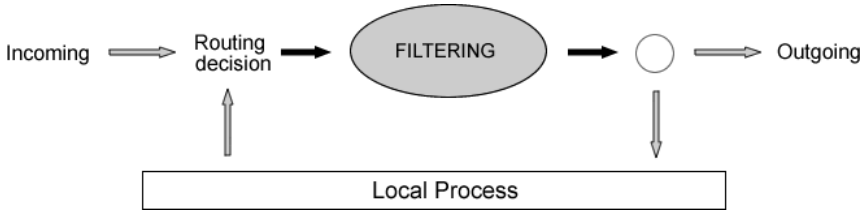



**Figure 2.4 –** Filtering model in this language

In order to be able to translate every filtering tables in the language and with this transformation, small changes are normally needed. The modifications needed for *Netfilter/iptables* are very light and are explained in the section 3.3.1.

To analyse distributed firewalls, DFA will need information about the network topology. We choose to store this in the firewall configuration file. In this way, the file contains the filtering information along with the network topology information. The entire file includes all the data to be able to understand all the rules when the firewall is isolated.

How to represent the location of the firewall in a network? Our solution is to define precisely each interface. For each interface, we list the IP addresses that can be reached by this interface. These lists of addresses are represented by the networks that can be reached through the interface. It should be noted that if a network associated with an interface is included in another network, then packets are considered as routed to the more specific network, as in a normal routing table. The four next tables (2.3, 2.4, 2.5, 2.6) define the structure elements used to define each interface. We explain how to fill the interface lists in details in section 3.2.3.

## 2.4 User-defined chains

In large networks, the number of rules written to implement a security policy can be very large. Moreover, such long rule lists can be complicated to understand. It is always interesting to be able

| **interface** | |
|---|---|
| *Description* | A physical or logical interface of the firewall. |
| *Attribute* | `name`: the name/identifier of the interface |
| *Allowed content* | `network`(+) |

**Table 2.3 –** Definition of the element `interface`

| **network** | |
|---|---|
| *Description* | Definition of an IP subnetwork. |
| *Attribute* | none |
| *Allowed content* | `ip`(1), `mask`(?) |

**Table 2.4 –** Definition of the element `network`

| **ip** | |
|---|---|
| *Description* | An IP address. |
| *Attribute* | none |
| *Allowed content* | An IP address in the format x.x.x.x ($0 \geq x \geq 255$) |

**Table 2.5 –** Definition of the element `ip`

| **mask** | |
|---|---|
| *Description* | Network mask associated with the IP addres. |
| *Attribute* | none |
| *Allowed content* | A mask address in the format x.x.x.x ($0 \geq x \geq 255$), not needed if equals to $255.255.255.255$. |

**Table 2.6 –** Definition of the element `mask`

to structure the rules, especially for the rules that share a common part. For instance, consider the rule set of Figure 2.5.

```
forwarded: prot=TCP & dst_ip=1.2.3.4 & dst_port=22 => ACCEPT
forwarded: prot=TCP & dst_ip=1.2.3.4 & dst_port=23 => ACCEPT
forwarded: prot=TCP & dst_ip=1.2.3.4 & dst_port=80 => ACCEPT

forwarded: prot=TCP & dst_ip=2.2.2.2 & dst_port=22 => ACCEPT
forwarded: prot=TCP & dst_ip=2.2.2.2 & dst_port=23 => ACCEPT
forwarded: prot=TCP & dst_ip=2.2.2.2 & dst_port=80 => ACCEPT

forwarded: prot=TCP & dst_ip=4.3.2.1 & dst_port=22 => ACCEPT
forwarded: prot=TCP & dst_ip=4.3.2.1 & dst_port=23 => ACCEPT
forwarded: prot=TCP & dst_ip=4.3.2.1 & dst_port=80 => ACCEPT

forwarded: dst_ip=LAN => ACCEPT
forwarded: => DROP
```

**Figure 2.5 –** A rule list without using user-defined chains

This example implements a policy where the only packets that can leave the network are those bound to the IP addresses 1.2.3.4, 2.2.2.2 or 4.3.2.1 and port 22, 23 or 80. Unfortunately, this policy requires a lot of redundancy in the rule set.

A solution to reduce this redundancy is to introduce user-defined chains. As explained in the first chapter and as the name indicates, chains permit to link several rules together. More precisely, a chain is a target that a rule can start from. To illustrate this concept, Figure 2.6 corresponds to the previous policy formulated with chains.

```
forwarded: prot=TCP => tcp
tcp: dst_ip=1.2.3.4 => trust_srv
tcp: dst_ip=2.2.2.2 => trust_srv
tcp: dst_ip=4.3.2.1 => trust_srv
trust_srv: dst_port=22 => ACCEPT
trust_srv: dst_port=23 => ACCEPT
trust_srv: dst_port=80 => ACCEPT
forwarded: dst_ip=LAN => ACCEPT
forwarded: => DROP
```

**Figure 2.6 –** The same rule list as 2.5 but using user-defined chains

This representation is simpler and shorter than the previous one. It is possible to recover the long list by linking the rules together. Figure 2.7 represents the nested chains in this example. It can be noticed that a given chain can be called several times. Of course, no chain loop has to appear in the rule definition!

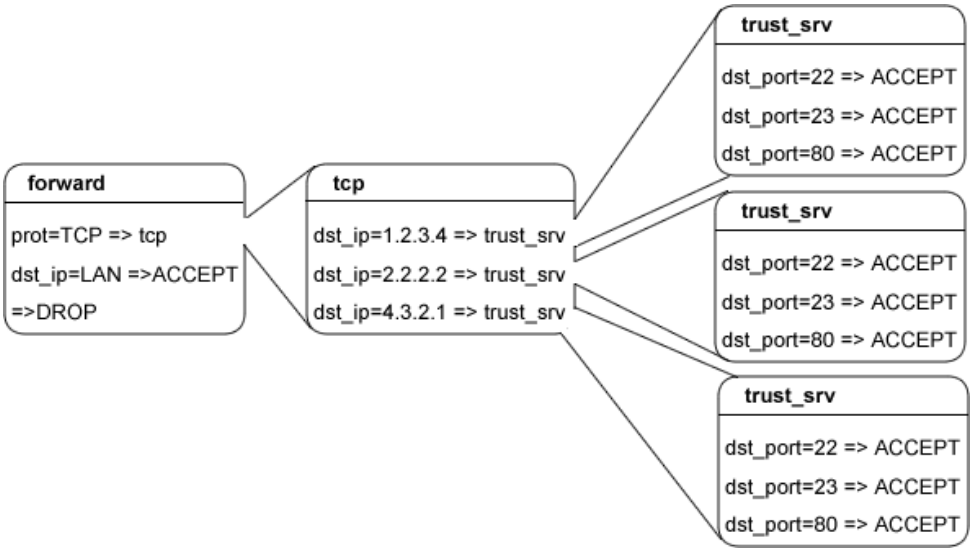



**Figure 2.7 –** Structure of the chains in the example of Figure 2.6

*Netfilter* products are famous for the use of this concept of chain. First, there are some built-in chains: INPUT, FORWARD and OUTPUT that match incoming, forwarded and outgoing packets. Moreover, users have the ability to define new chains. These user-defined chains can be useful in some case as explained previously.

Chained rules are useful to simplify the definition of the rules but not for their representation. A simple way to represent a rule set is to do it with a tree where each node represents a field choice. Figure 2.8 provides the tree representation of the rules shown as example just before. Unfortunately, it can be very difficult to build such trees with chains. Imagine a rule set :

```
fwd: prot=TCP & dst_port=0-1024 & src_ip=1.0.1.0/24 => chain1
[...]
chain1: dst_port=80 => chain2
[...]
chain2: dst_ip=1.2.3.4 & src_ip=1.0.0.0/8 => ACCEPT
[...]
```

The only way to insert the rule resulting from this chain in a tree is to merge it into a single rule. An insertion with a linear reading of these chains is nearly unfeasible.

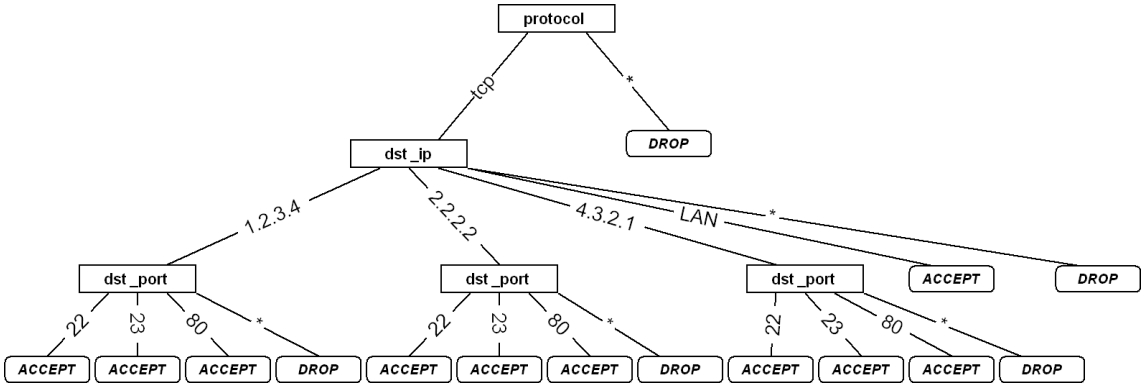



**Figure 2.8 –** Tree that represents the set of rules of Figure 2.6

Therefore, to simplify the further work, we have decided to merge the chained rules into simple rules. This is the only solution to remove chains from languages that use user-defined chains.

## 2.5 Rules and fields for filtering

Now that chains have been introduced, we can add the other elements of the language structure. The first one is the basis element of a filter table, a rule. A rule contains a target and a list of conditions. If all conditions are matched by a packet, it is sent to the target. An important point to notice is that all the conditions to match in a rule are linked with an AND operation. In some languages, rules may occasionally have some hidden disjunctions. In our language, it is authorized to match «source port between 20 and 23» since it can be written in one element. However, a rule containing «source port 20 or 23» has to be split in two rules. The definition of a rule in the language is shown in Table 2.7.

The reason why the cardinality of `interfacein`, `interfaceout` and `protocol` is "zero or more" and not "zero or one" is because it is needed when the fields are inverted. For instance, the rule `prot!=tcp & prot!=udp => ACCEPT` needs two `protocol` elements since only one value can be contained in an XML `protocol` element.

| **frule** | |
|---|---|
| *Description* | A rule of the firewall filtering table that matches some packets to give them a specific target. The children are linked with an AND operation. |
| *Attribute* | `target`: the target whose matching packets are sent to. (ACCEPT\|DROP) |
| *Allowed content* | `interfacein`(*), `interfaceout`(*), `protocol`(*), `ipv4`(?), `tcp`(?), `udp`(?), `icmp`(?), `ah`(?), `esp`(?), `established`(?) |

**Table 2.7 –** Definition of the element `frule`

The next elements correspond to the fields that define the rules. For most elements, the attribute `inverted` is added. If it is false, the packets matched are those with the value of the element. On the opposite, if it is true, the packets are matched if their values are different from the value of the element.

The field elements are arranged in groups when they cover the same topic. For instance all matches about IPv4 rules are gathered together inside an `ipv4` element.

The basic rules are about packet input and output interfaces and are defined in tables 2.8 and 2.9.

| **interfacein** | |
|---|---|
| *Description* | The input interface of the packet. |
| *Attribute* | `inverted`: `true` if only packets without this value are matched, `false` if only packets with this value are matched. |
| *Allowed content* | A name of an interface defined in one of the `interface` elements. |

**Table 2.8 –** Definition of the element `interfacein`

| **interfaceout** | |
|---|---|
| *Description* | The output interface of the packet. |
| *Attribute* | `inverted`: `true` if only packets without this value are matched, `false` if only packets with this value are matched. |
| *Allowed content* | A name of an interface defined in one of the `interface` elements. |

**Table 2.9 –** Definition of the element `interfaceout`

Next, the `protocol` field is described in table 2.10. Note that the value used in this field is part of the the keywords defined by the IANA in lower case. A list of these values can be found in the `/etc/protocols` file in Linux systems or on the IANA webpage about protocol numbers[1].

| **protocol** | |
|---|---|
| *Description* | The protocol used in the packet. |
| *Attribute* | `inverted`: `true` if only packets without this value are matched, `false` if only packets with this value are matched. |
| *Allowed content* | A valid name of protocol as defined by IANA and in lower case. |

**Table 2.10 –** Definition of the element `protocol`

[1] IANA protocol numbers: `http://www.iana.org/assignments/protocol-numbers`

Tables 2.11, 2.12, 2.13 and 2.14 describe the matches about IP version 4. The role of the `ipv4` tag is to aggregate the IPv4 matches. The `ipv4` element is not needed if the group is empty. The `ip` and `mask` elements used are the same as those used in the `interfacein` and `interfaceout` elements (see tables 2.5 and 2.6).

| **`ipv4`** | |
|---|---|
| *Description* | Group of the rules about IP version 4. |
| *Attribute* | none |
| *Allowed content* | `src`(*), `dst`(*), `fragment`(?) |

**Table 2.11 –** Definition of the element `ipv4`

| **`src`** | |
|---|---|
| *Description* | The source IP address of the packet |
| *Attribute* | `inverted`: `true` if only packets without this value are matched, `false` if only packets with this value are matched. |
| *Allowed content* | `ip`(1), `mask`(?) |

**Table 2.12 –** Definition of the element `src`

| **`dst`** | |
|---|---|
| *Description* | The destination IP address of the packet |
| *Attribute* | `inverted`: `true` if only packets without this value are matched, `false` if only packets with this value are matched. |
| *Allowed content* | `ip`(1), `mask`(?) |

**Table 2.13 –** Definition of the element `dst`

| **`fragment`** | |
|---|---|
| *Description* | Match the fragments of IP packets. |
| *Attribute* | `inverted`: `true` if only the non fragmented packets and the first part of fragmented ones are matched, `false` if only the second, third, ... fragments are matched. |
| *Allowed content* | none |

**Table 2.14 –** Definition of the element `fragment`

TCP elements look like the IP ones except that the matched fields are obviously different. Tables 2.15, 2.16, 2.17, 2.18 and 2.19 are about TCP.

| **`tcp`** | |
|---|---|
| *Description* | Group of the rules about TCP. |
| *Attribute* | none |
| *Allowed content* | `srcport`(*), `dstport`(*) |

**Table 2.15 –** Definition of the element `tcp`

UDP is very similar to TCP. It is defined in table 2.20 and uses the same child elements.

| **srcport** | |
|---|---|
| *Description* | The source port of the packet for this protocol. |
| *Attribute* | `inverted`: `true` if only packets without this value are matched, `false` if only packets with this value are matched. |
| *Allowed content* | The port value if there is only one value or `from` and `to` for a range. |

**Table 2.16 –** Definition of the element `srcport`

| **dstport** | |
|---|---|
| *Description* | The destination port of the packet for this protocol. |
| *Attribute* | `inverted`: `true` if only packets without this value are matched, `false` if only packets with this value are matched. |
| *Allowed content* | The port value if there is only one value or `from` and `to` for a range. |

**Table 2.17 –** Definition of the element `dstport`

| **from** | |
|---|---|
| *Description* | The lower-bound of a range |
| *Attribute* | none |
| *Allowed content* | A positive integer |

**Table 2.18 –** Definition of the element `from`

| **to** | |
|---|---|
| *Description* | The higher-bound of a range |
| *Attribute* | none |
| *Allowed content* | A positive integer (has to be higher than the value of its "brother" `from`). |

**Table 2.19 –** Definition of the element `to`

| **udp** | |
|---|---|
| *Description* | Group of the rules about UDP. |
| *Attribute* | none |
| *Allowed content* | `srcport`(*), `dstport`(*) |

**Table 2.20 –** Definition of the element `udp`

ICMP type and code can be also configured, see tables 2.21 and 2.22. An ICMP packet has always a message type and no payload. Some of these messages may also have a specific code associated to the type. For instance: *Echo reply* ICMP has the type 0, this type has no subcategory. However, the *Destination Unreachable* ICMP (type 3) uses 16 codes. These codes and types can be found on the IANA definition of ICMP type numbers[2].

The last protocols supported by the current version of the language are those used by IPsec: ESP and AH. These two protocols rely on the concept of key-id which is transmitted in the packet header. This is called the SPI (Security Parameter Index) and is the only match that can be configured in the language for these protocols. Details can be found in tables 2.23, 2.24 and 2.25. More complex firewall configurations about IPsec are possible [Hame 05] but are not widely used today.

[2]IANA definition of ICMP type numbers: `http://www.iana.org/assignments/icmp-parameters`

| **icmp** | |
|---|---|
| *Description* | Group of the rules about icmp. |
| *Attribute* | none |
| *Allowed content* | `type`(*) |

**Table 2.21 –** Definition of the element `icmp`

| **type** | |
|---|---|
| *Description* | The ICMP type and code if any |
| *Attributes* | `inverted`: `true` if only packets without this value are matched, `false` if only packets with this value are matched.<br>`value`: the ICMP type number.<br>`code`(optional): the code associated with this ICMP type |
| *Allowed content* | none |

**Table 2.22 –** Definition of the element `type`

It should be added to the language in a next version.

| **ah** | |
|---|---|
| *Description* | Group of the AH rules. |
| *Attribute* | none |
| *Allowed content* | `spi`(*) |

**Table 2.23 –** Definition of the element `ah`

| **esp** | |
|---|---|
| *Description* | Group of the ESP rules. |
| *Attribute* | none |
| *Allowed content* | `spi`(*) |

**Table 2.24 –** Definition of the element `esp`

| **spi** | |
|---|---|
| *Description* | The Security Parameter Index of the packet. |
| *Attribute* | `inverted`: `true` if only packets without this value are matched, `false` if only packets with this value are matched. |
| *Allowed content* | The SPI if there is only one value or `from`(?), `to`(?) for a range. |

**Table 2.25 –** Definition of the element `spi`

## 2.6 IP flows

Very effective policies used in firewalls are built around the concept of data flows. It is common to see policies like « *accept the connection initiations from the inside to the outside and deny the others* ». Given that a unidirectional connection generates packets in both directions, a firewall

cannot block those coming from the outside. The issue is also to find which host has initiated a data flow in order to reject unauthorized packets.

There are two main ways to achieve that. The first one is to implement a finite state machine in the filtering software and is called *stateful*. The second is to interpret the flags of the packet header and is called *stateless*. Let us start with a quick review about the protocols above IP.

### 2.6.1 Data flows with TCP, UDP and ICMP

First, a flow is defined as the set of all packets from one computer to another, relating to an instance of a service and during a given period of time. A part of these general concepts is based on [Zwic 00].

#### TCP

TCP is the protocol that is the most widely used for reliable communication over the Internet. A successfull TCP connection between two hosts ensures that the data has been correctly transmitted to the destination, in the same order as the packets were sent and without duplication. If one of these conditions is not fulfilled, the connection is broken and an error is notified to the upper layers.

Even if the data is only transmitted in one direction as in a FTP transfer, packets have to be exchanged in the both directions. Acknowledgment packets are sent in the opposite direction of the data packets to ensure the previous properties. So the TCP connection cannot be initiated if packets are not authorized in both directions.

One of the more important issues for TCP packet filtering is to identify which host has initiated the connection. This analysis can be done by looking at the packet headers. Important header information is, of course, the source and destination ports but also the TCP flags. TCP flags whose we care about are `syn` and `ack` for the beginning and `fin` and `rst` for the termination.[3]

A TCP connection is initiated by the *three-way handshake* (except for simultaneous open). Figure 2.9 represents the values of TCP flags during the handshake. If the implementation conforms to RFC [Post 81], a connection could not be established if the flags are not well-defined. The first two packets have the `syn` flag set, this flag is only used for initialisation. The first packet has the particularity that it must have its `ack` bit not set. This characteristic is important since it allows to distinguish it with the second one. From the third one, the packets cannot have the `syn` bit set. They also have the `ack` bit set to notice that the acknowledgement field contains the number of the next expected piece of data. Of course, all these properties are useful to detect the start of a connection.

The flags `rst` and `fin` are used to close a TCP connection. The `rst` is sent for an abrupt close, i.e. the connection is immediately closed when a packet with this flag is received or sent. The `fin` bit is used to perform a graceful shutdown of the connection. Each host sends a packet with this flag to the other to say goodbye. Note that `fin` segments are acked. Packet filtering can use these flags to consider the connection as closed and to refuse subsequent packets.

[3] Flags and other information about TCP can be found in RFC[Post 81].

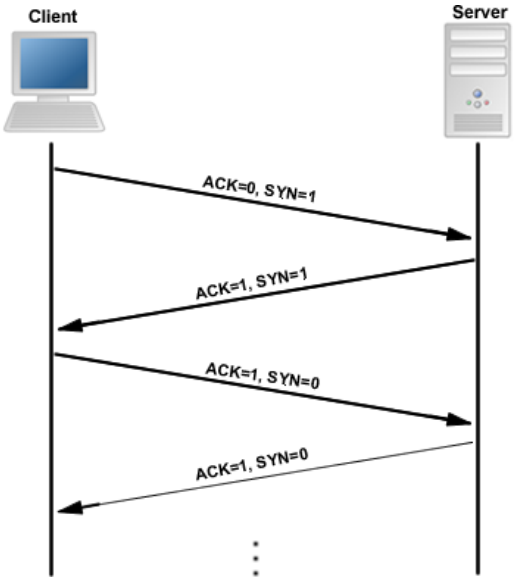



**Figure 2.9 –** `Ack` and `syn` flag in TCP three-way handshake

### UDP

In contrast to TCP, the User Datagram Protocol (UDP) is a non-reliable protocol. An application using UDP has to make its own data control if necessary. There is no guarantee that datagrams sent are actually received by the destination or are received only once. The benefit of UDP is the low overhead due to the absence of all the control mechanisms.

UDP packet headers contain, like TCP, the source and destination ports. However, for packet filtering, UDP is more difficult to trace. Moreover, it does not contain any flag or sequence number. Thus it is not possible to know for sure that a packet is the first one of a flow or a response. A way to determine data flows is to suppose that a packet sent with the same addresses and ports as a previous one is probably from the same flow.

### ICMP

ICMPs are messages sent in IP packets and contain status and control messages. Well-known ICMP messages are *echo request/reply*, *time exceeded*, *destination unreachable* and *redirect*. The header of such messages does not contain port information but contains its type. The filtering can be done on this field and even on extra-options proper to the type of ICMP.

There is no pure ICMP flow to observe most of the time. Only four types of ICMP messages generate an ICMP response[Andr]: *echo request/reply*, *timestamp request/reply* (deprecated), *information request/reply* and *address mask request/reply*. Since there is always at most an unique reply to a request, ICMP flows can be considered as finished when the reply has been received.

Actually most of the ICMP messages are sent in reply to TCP or UDP packets. That is why ICMP messages have to be considered as special responses to an existing data flow. Some of these messages are terminal error messages concerning TCP or UDP connections. The *net unreachable* and *net prohibited* ICMP messages are two examples of this, the client does not receive anymore packets after these ICMP messages. When one of those is detected, the firewall can consider the

TCP or UDP connection as finished. This is how the finite state machine of *Netfilter/iptables* is implemented. [Andr]

### 2.6.2 Finite state machines

Firewalls that implement a finite state machine to control the data flows are called stateful firewalls. These firewalls simply keep track of each connection thanks to the source and destination addresses and ports. The advantage of this solution in comparison with the next one is that all packet types can be analysed, even UDP and ICMP. The following paragraphs describe a simple implementation.

The memory of a firewall state machine can be viewed as a list of entries, one for each connection. A typical entry is composed of the protocol of the flow followed by local address, local port, remote address and remote port, and a countdown. For instance: «`udp 1.1.1.1 32000 2.2.2.2 80 60`»

When a packet reaches the firewall, the system controls whether the packet flow is registered in the entry list. If so, the countdown field is reset to its initial value. Otherwise, a new entry is created if the connection is authorized. Then, the subsequent packets received for this flow will reset the countdown at their arrival. The countdown is decremented each second; if it reaches zero, the corresponding entry is removed.

This simple implementation is sufficient to observe the policy "accept the connection initiations from the inside to the outside and deny the others". To implement it, new entries can only be created when a packet is leaving the network.

More complex implementations are possible. The *Netfilter/iptables* implementation is detailed in [Andr]. In *iptables*, four different states are used for each flow: *new*, *established*, *related* and *invalid*.

- The *new* state matches only when a packet is the first one of a flow and an entry has just been entered in the "state table"[4].
- The *established* state is put in the state table when packets of a flow have been seen in both directions. When the *established* state is the only one allowed, it means that packets can be sent only in reply from an authorized request in the other direction.
- *Related* is used to authorize a flow that is related to another flow. Common examples of such connections are FTP-data flows. ICMP errors are also considered as *related*. We will not enter into details since this is not important for the purpose of the problem.
- The *invalid* state means that the packets do not have any state or that they cannot be identified. Such packets are usually dropped.

An important point to notice is that, by default, `iptables` does not look at all at the TCP flags. This is a choice of *Netfilter* to improve dynamic changes in firewalls[5].

[4] In *Netfilter*, the connection tracking is done by a special framework inside the kernel called `conntrack`.
[5] See the section "B.2. State NEW packets but no SYN bit set" in [Andr] for more information.

### 2.6.3 Flag analysis

An alternative exists to this complex state machine; this is to simply look at the flags inside the packet header. As seen in the section 2.6.1, only TCP has a header that can be interpreted. To realize the policy « accept connections from inside to a remote host and deny connections from the outside », two solutions are used in packet filtering implementations. The first one is simply to not accept TCP packets from the outside without the `ack` bit set. The flag `ack` cannot be set in the first packet of the three-way handshake, so the connection cannot be established. The *Cisco* implementation examines this bit when the word `established` is specified[6][Cisco]. The second solution is to use the `syn` bit. *Netfilter* uses it to define the match «`--syn`» that means `SYN=1,ACK=0,RST=0` : this flag configuration is only used in the first packet of a TCP connection. Using «`!--syn`» on the input link will drop connection attempts from the outside. The only difference is that, in the second solution, a reply packet is accepted if `syn=0`, `ack=1` or `rst=1`. Packets with flags `syn=0,ack=0` or `ack=0,rst=1` will be accepted by the second solution and not by the first one. However, this difference is unimportant.

Note that it is possible with *iptables* to match any combination of flags but this does not seem to be very common.

### 2.6.4 Stateful versus stateless firewalls

Two different solutions are possible to filter connection requests. To translate these rules in our meta-language, it is interesting to see if there is an actual difference between these two approaches.

The state machine and the TCP flag analysis are mainly used to accept or reject connection requests. For this reason and to simplify the intermediate language, only these matches are kept. Unfortunately, custom flag combinations and `RELATED` matches cannot be translated in the language.

In the direction of authorized connections, the following matches are commonly accepted (i.e. are associated with an ACCEPT policy in rules): «`NEW, ESTABLISHED`» for a stateful firewall and nothing for TCP flag analysis. Actually, «`NEW, ESTABLISHED`» does not filter anything but it initiates the state machine. Then the filtering is the same in both cases: nothing is done. In the limited direction, the encountered matches are «`ESTABLISHED`» for state machine and «`!syn`» for flags. The `ESTABLISHED` effect is nearly the same as the «`!syn`» one, it rejects new TCP connections. More precisely, it is not exactly the same from a security point of view but it is on the semantic one.

Only the semantic is important in our language, so the concept of connection can be represented in the language by a single element. The table 2.26 contains the information about this element.

| **`established`** | |
|---|---|
| *Description* | Matches packets belonging to a known flow. |
| *Attribute* | `inverted`: `true` if packets has to be the first one of flow, `false` if it has to be a response to a existing flow. |
| *Allowed content* | none |

**Table 2.26 –** Definition of the element `established`

[6] To be precise, *established* is matched if the TCP header has the `ack` or `rst` bits set.

## 2.7 Extensions

One of the main objectives of our language was to be as extensible as possible. Extending the language could be needed for several reasons:

- The current version has – above all – been built to parse the main rules of *iptables*. It does not mean that other firewall implementations cannot be translated into our language. These other firewall languages might force us to add new matches corresponding to new fields supported.
- As indicated before, a lot of extensions are available for *iptables* [Netfilt, Andr]. Some of them cannot be expressed in the current language. Thus, our language could be extended to express some new matches for these extensions.
- Most of the current filtering implementations only consider IPv4 as the network layer protocol. In a few years, IPv6 will certainly be used worldwide and filtering tables will have to conform.

To add a new match in the language, the only thing to do is to add the new elements at the right place with the same structure. General purpose elements are put as children of `frule` and protocol specific properties are grouped in elements with the same name as the protocol. For example, the only thing to do to support IPv6 is to add an `ipv6` element as a new group in our language. In this group should be added all the properties of IPv6 that can be matched. A new IP address element obviously has to be defined.

The `filtertable` element is a child of the root element and contains all the filtering rules. If a NAT table has to be defined later, this table could be placed next to the filtering table without mixing filtering rules and NAT ones.

## 2.8 Conclusion

The language introduced in this chapter fulfils the objectives that have been set. The language is expressive enough to represent most of the matches that may appear in firewall configurations. Furthermore, the language can be easily extended to support future matches. The meta-language used to represent our language is XML.

Moreover, the construction of our language was the opportunity to take some global important decisions for the rest of the thesis:

- Only the filtering rules are taken into account in the firewall configurations. Log- and NAT-rules are ignored.
- The network interfaces of the firewall are directly defined in the configuration file of our language.
- The packets coming from and going to the host itself are considered as coming from or going to a virtual interface that represents all the addresses of the firewall host.
- The user-defined chains do not exist in our language. Rule merging is thus needed to remove these chains from the imported configurations.

- The stateful and stateless flow detection rules are all translated into the `established` element that matches the flows that are already established.

Chapter 3

# The parser

This third chapter deals with the implementation of the parser. This parser has been designed to translate configurations of common firewall configuration files in the language defined in the previous chapter. In the current version, the only language supported is *Netfilter/iptables* but the parser has been designed to be easily extended to support other configuration languages. The first section describes how to create and parse XML files in Java. Secondly, the general parsing mechanism and the class organization are introduced. Next, the parsing of *iptables* rules is explained in more details. Finally, the way to extend the implementation to other languages is described.

## 3.1 XML and Java

As seen in the previous chapter, XML is used to define well-structured documents. Writing and reading an XML file is independent from its structure and can be performed in several ways. Two main approaches exist: DOM and SAX.[Haro 02a]

### 3.1.1 The Document Object Model (DOM)

DOM is an API for accessing and manipulating XML documents as tree structures. It has been designed by the W3C[W3C] and uses a hierarchical arrangement of objects to represent the XML entities.

DOM is appropriate when XML documents have to be browsed and changed. It is easy with DOM to add and delete elements. Of course, the drawback is that the entire document has to be parsed before it can be manipulated. Moreover, such a tree structure may consume lots of memory.

### 3.1.2 Simple API for XML (SAX)

SAX is an event-based API. This means that a SAX parser reads linearly the XML files and reports the elements encountered. In opposition to DOM, SAX does not use a lot of resources and therefore can manipulate very large files without any problem. However, the manipulation of XML documents with SAX is much more complex.

### 3.1.3 DOM or SAX?

In DFA, the firewall configuration structure is built step by step. The rule elements are built as trees before being inserted in the firewall structure. Then, a dynamic tree structure is needed to build the entire document.

We choose to use JDOM to parse the XML files. JDOM is a Java API designed to take advantage of the Java language to obtain better tree representations of XML documents. JDOM has less functionality than DOM but is much easier to use for the kind of manipulations needed in this work. JDOM can equally use a SAX Parser to build its document structures though they are still structured as trees. More information about JDOM can be found on the project webpage[1].

To make clearer the class descriptions of the next sections, here are the main classes of JDOM used by our implementation:

**Document:** Class that contains the root of the document XML tree. It can be easily outputted to an XML file.

**Element:** Class that represents an XML element as defined in the previous section. An element is identified by a name and can have a string text value, some child elements and some attributes.

There are many more JDOM classes but these few classes are sufficient to manipulate the entire XML tree. Nearly everything is done by the `Element` class. Its methods allow to obtain the value stored, the value of an attribute with a given name and its child elements. More information about XML with Java and JDOM can be found in [Haro 02b, McLa 01].

## 3.2 Common implementation

This section describes the part of the implementation that does not depend on the input language. It explains the class structure of the application and general parsing mechanisms. The graphical user interface for the parser part will not be explained in this chapter; it provides a simple interface to select input and output files for parsing.

### 3.2.1 The main classes

The application is hierarchically organised as traditionally in Java applications. A general class that represents the firewall configuration calls a parser to build the XML document. This parser finds string rules in the input file and creates `Rule` elements associated with those. Then, the rules extract information found in their string representation and create an XML structure from it. Figure 3.1 represents the basis classes of the parsing part of the application and the *iptables*-specific classes. The « import » links correspond to a use (or a call) of a class by another one. It may be noticed that language-specific classes extend the parser and the rule ones in order to define the operations particular to the parsed language.

[1] The JDOM project website: `http://www.jdom.org/`.

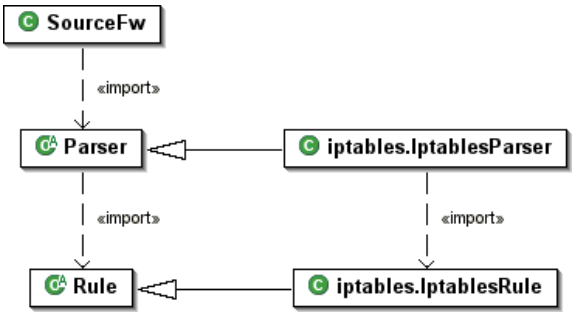



**Figure 3.1 –** Diagram of the parser main classes

#### `SourceFw` class

The `SourceFw` class is the representation of the imported file. This class manages:

1. the opening and closing of the input and output files;
2. the exception handling: it prints the error messages in the message stream displayed on the GUI;
3. the call of the parser class of the suitable language when parsing is requested.

This class is the one called by the GUI to control parsing. The implementation is quite simple and does not need more explanation.

#### `Parser` class

The `Parser` class is an abstract class. In Java, an abstract class is a class that cannot be instantiated. Such a class defines one or more abstract methods. An abstract method has no definition i.e. is not implemented. So, such a class has to be extended with classes that must extend at least the abstract methods of their super class [Lewi 04]. In our application, `Parser` is an abstract class that must be extended with language-specific parsing classes. In this way, the methods that have language-specific parts can be implemented in their class. On Figure 3.1, `IptablesParser` extends `Parser`, i.e. it defines the *iptables* specific methods.

Here are the various functions of the `Parser` class and its child classes:

1. It parses the input file to find the relevant rule lines and stores them a list;
2. It calls the parsing method on each rule; it also suppresses and duplicates rules if needed;
3. It imports the list of interfaces given by the user (and generates an empty one if needed);
4. It returns the structure of the entire XML document.

The first and second steps are implemented in the language-specific classes. The third one is explained in section 3.2.3. The last one consists in the creation of a `firewall` element, the root of the XML tree. Then, the `interface` elements have to be added to the root as well as the `filtertable` element. Finally, all the rules of the list are added as children of `filtertable`.

#### `Rule` class

At its creation, a rule is only a string representation of a filtering rule; this string can be parsed afterwards. The `Rule` class extends the `org.jdom.Element` one since it is actually a `frule` element (seen in the previous chapter in Table 2.7). In this way, when a rule parses its string representation, sub-elements are created according to the matches and are attached to the rule. When parsed, a rule can therefore directly be inserted into the XML tree structure.

The `Rule` class is also an abstract class. Indeed, the parsing of a string rule to its XML representation depends on the language parsed.

### 3.2.2 The rule child classes

The children created during parsing belong to one of the following classes: `Protocol`, `Established`, `IntefaceIn`, `InterfaceOut`, `Ipv4`, `Tcp`, `Udp`, `Icmp`, `Ah` or `Esp`. All these classes extend `org.jdom.Element` since each instance of them is a child element of a rule element in the programming structure as well as in the XML structure. The XML name of these child elements corresponds to the names of fields defined in the previous chapter. So when a rule parses its string representation, it builds its XML tree.

Figure 3.2 represents the classes that can be child of the class `Rule`. Note that some classes have other classes as child too. It corresponds to the field groups explained in the previous chapter. When the class box is contained into another class box, it means that it is an inner class.

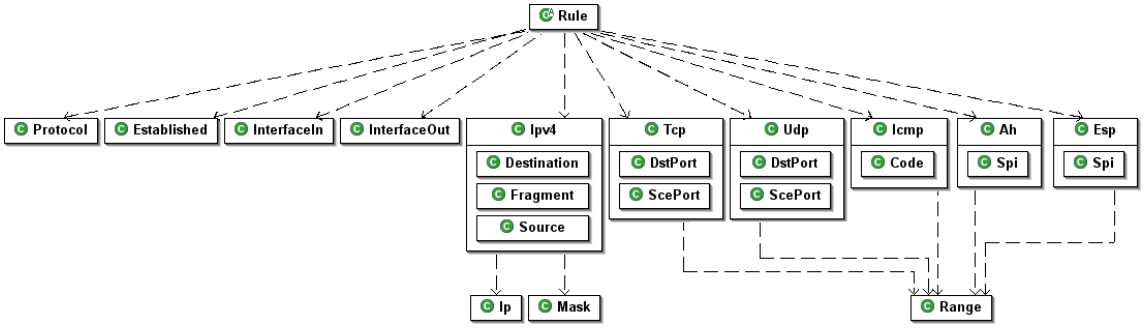



**Figure 3.2 –** Class diagram of the Parser package

### 3.2.3 The interface definitions

As explained in the previous chapter, we chose to include network information in the firewall configuration files. So, all the network interfaces (or virtual interfaces) must be precisely defined. The preferred approach to thoroughly describe the network interfaces of a firewall is to provide all the IP addresses that can be reached by each interface. The way to do it is to supply an XML interface file with the definition of each interface of the firewall. In order to not have to create this file from scratch, a feature has been added to the application to generate an empty XML structure for this file. It looks for the interfaces used in the parsed rules and creates an empty skeleton.

How to fill it? The first idea is to add for each interface the address of the network this interface belongs to. These networks can be found in the listing given by the network status listing

(`ifconfig` under Linux). Unfortunately, as illustrated at Figure 3.3, in some configurations, other networks can be joined behind an interface. In this example, the network $5.5.5.0/24$ can also be joined by the interface `eth1`. So, supplying $1.2.0.0/16$ is not sufficient.

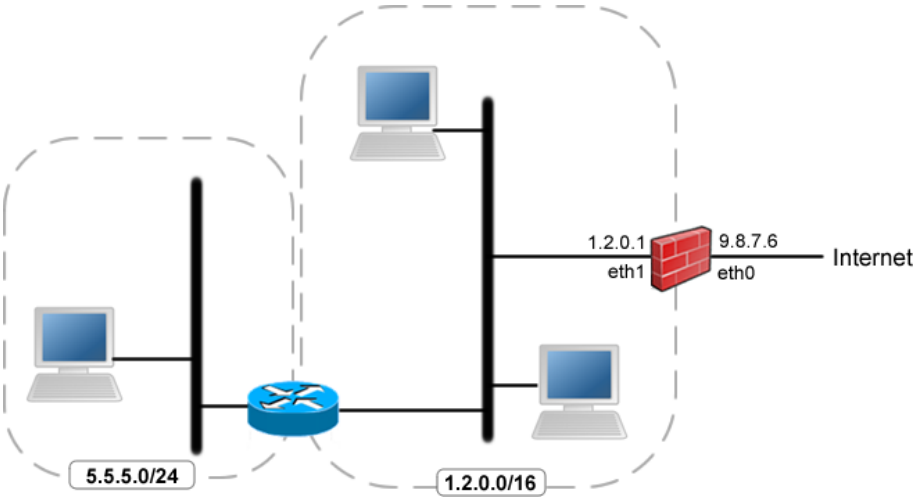



**Figure 3.3 –** A typical network where the firewall interface definition has to be made

Another idea to fill the interface list is to examine the routing table of the host. This table gives all the reachable networks and the interface used to join them. However, in some hosts (in some bridges for instance), the routing is not done at the same level and the routing table does not provide any interesting information. Anyway, an unfailing way to fill the interface list is to know the network topology and to express it directly.

Because of the lack of time, no graphic interface has been done to manage the interface list. In the current version of the application, a skeleton of the interface list with the interfaces discovered during parsing can be exported. Then, the file has to be filled manually. When this has been done, DFA is able to generate the entire firewall configuration file. The structure of the interface list is the following: a root element `interface_list` and as children, the `interface` elements defined in the previous chapter (table 2.3).

For instance, the interface list has to be filled as shown in Figure 3.4 for the network represented at Figure 3.3. As explained in the language definition, the interface `firewall_host` is used for all the packets from and to the firewall itself.

It is possible that an actual interface does not appear in the skeleton to fill. This may occur if the interface is not contained in a rule of the filtering table. Nevertheless, it is important to add it to the interface list in order that the packets from this interface are not associated with another one.

It can be noticed that the *loopback* interface does not appear. Actually, it is absolutely essential that it is removed from the list. Indeed if it is kept when the two firewalls are analysed, the address 127.0.0.1 in the two configurations will refer to addresses that do not correspond to same physical host.

```
<?xml version="1.0">
<interface_list>
  <interface name="firewall_host">
    <network>
      <ip>1.2.0.1</ip>
    </network>
    <network>
      <ip>9.8.7.6</ip>
    </network>
  </interface>
  <interface name="eth0">
    <network>
      <ip>0.0.0.0</ip>
      <mask>0.0.0.0</mask>
    </network>
  </interface>
  <interface name="eth1">
    <network>
      <ip>1.2.0.0</ip>
      <mask>255.255.0.0</mask>
    </network>
    <network>
      <ip>5.5.5.0</ip>
      <mask>255.255.255.0</mask>
    </network>
  </interface>
</interface_list>
```

**Figure 3.4 –** Interface list of the firewall in Figure 3.3

## 3.3 Dealing with *iptables*

*Iptables* is the packet filtering implementation used in Linux kernels 2.4 and 2.6. Its functionalities have been described in section 1.1.3. This language is the first one supported by the parser; here are several reasons of this choice:

- Linux systems can be found both in gateways and in end-computers. Moreover, this free operating system is widely used in servers and in routers. So, *iptables* is a common tool for packet filtering in networks.
- Several *iptables* filtering tables of a large network were available. It is very important to make the validation of this work.
- *Iptables* is a rather complex implementation. If the parsing is performed without any problem with *iptables*, other language parsers can probably be implemented.

As seen in the previous section, the main classes `Parser` and `Rule` are extended by classes proper to *iptables*. These classes implement the entire parsing of the *iptables* output. They have also to deal with *iptables*-specific problems like the user-defined chains. This section provides an overview of all the processing from the input file to the XML rules. Although it relates to *iptables*, the parsing modus operandi would be very similar for other languages.

### 3.3.1 Parsing of the input file

This section discusses the parsing viewed from the angle of the `Parser` class. It relates to the rules in general but not to the parsing of these rules that is the subject of the next section.

#### Choice of the input file

The first problem is to select which output of *iptables* to parse. An *iptables* rule table is built by using the `iptables` command to add all the rules one by one. For outputting the entire table, several solutions exist. The following discussion is inspired by the same problem resolved in another thesis about *iptables* [Scha 04].

The `iptables` command called with the parameter `-l` returns a structured table that represents all the rules entered. This table is useful for a human to have a quick overview on the configuration but it is not easy to parse though. Another possible output for *iptables* is the one given by the `iptables-save` command. The result is a linear listing of rules with approximately the same structure than the rules entered previously. This listing is more difficult to quickly understand for a human but it is much easier to parse for an application. Moreover, `iptables-save` does absolutely output regular rules in spite of the synonymous terms to define the same matches. This last point is interesting because whether you enter `-s 1.1.1.1`, `-src 1.1.1.1` or `-source 1.1.1.1` to indicate that you want to match the packets with 1.1.1.1 as source IP, the `iptables-save` output will always be `-s 1.1.1.1`. So, it simplifies the parsing by using a shorter vocabulary.

#### Structure of the input file

`iptables-save` has a quite regular output: a simple example is shown in Appendix A.1. First, comment lines begin with a `#`. These lines can obviously be ignored. Next, the output is divided into three parts: *filter*, *nat* and eventually *mangle*. Each part begins with a line `*<part-name>` and finishes with `COMMIT`. As explained previously, only the *filter* part is kept for parsing. Each of the three parts has the same general structure. First of all, each chain is defined (one per line). The structure of these lines is the following:
`:<chain-name> <chain-policy> [<packet-counter>:<byte-counter]`
These lines are parsed to build a list of the chains used in the table and to know the default policy for basic chains. Only INPUT, OUTPUT and FORWARD have a default policy. The way to deal with it is explained below. When the chain definitions have been handled, filtering rules remain. Each rule line is used to create a new `Rule` object that is put in a rule list.

#### Default policies

As explained in the previous section, the `iptables-save` output contains the default behaviour of the main chains. However, as noticed in the section 2.1.2, the XML intermediate language does not support default policies, each packet has to be matched by at least one rule. So, the solution to observe this rule is to convert these default behaviours into rules. For instance, DROP as default policy for *forward* chain is equivalent to a rule with no field (matches all packets) whose target is DROP: `forward: -> DROP`. Then, such rule is added at the end of the rule list for each default policy; there is no loss of information with this procedure.

### Replacement of the INPUT and OUTPUT chains by a new interface

In the description of the firewall language, it was explained that all the packets passing through the firewall box are forwarded ones. The problem is that some packets are going to or are native from the host itself. The solution suggested was to consider that this host can receive and send packet through a virtual interface named `firewall_host`.

Let us add this interface now. Rules must be modified so that no change of policy was noticed. In *iptables*, filtering tables have three built-in chains: FORWARD, INPUT and OUTPUT. Rules of FORWARD chain do not refer to packets from and to the `firewall_host` interface. So, a condition «interfacein $\neq$ `firewall_host`» and «interfaceout $\neq$ `firewall_host`» has to be added to all the rules starting from the FORWARD chain. This is done by adding a rule «`-i ! firewall_host -o ! firewallhost -j FORWARD`». Note that it is not a valid `iptables-save` rule since it does not have a input chain. In our implementation, that rule means that it is a root rule, i.e. all packets are concerned by this rule. In the same way, the rules: «`-o firewall_host -j INPUT`» and «`-i firewall_host -j OUTPUT`» are added for packets respectively to and from the firewall itself.

### Parsing of all the rules

As explained in the parsing of the input file, rules are stored in a rule list when the parsing of the file is finished. Actually each rule is not parsed yet, it is just an object that contains a string representation of the rule. Then, all the rules are therefore parsed one by one. The way of parsing the rule is explained later but the result of the parsing is one of the following:

1. The rule will never be matched by a packet or the rule does not send the packet matched in a definitive target (it is the case if the target is LOG for instance).
2. All the rule matches are not linked with a conjunction (AND) and then the rule has to be split in two or more rules. The *multiport* match for example permits to give a list of ports to match. If the ports given are «$3, 4, 6$», the rule cannot be written as a conjunction.
3. The rule is parsed without problem.

In the first case, the rule must simply be dropped. In the second one, the rule is split and each of these new rules resulting of this splitting are added at the position of the old rule (the order between the rules has to be observed). Finally, if everything is all right, the rule is kept in the list.

### *Iptables* chains

As explained in the section 2.4, we have decided to remove all the user-defined chains of the configuration file of *iptables*. This is not an easy job.

Figure 3.5 shows the simplified pseudo-code of the algorithm used by the *iptables* parser to make the chains disappear. The list of rules is called `rules`. `fromchain` is the chain related to a rule and `target` is the target of a rule. At the beginning of the algorithm, `allsources` is a hashtable that contains for each chain the list of rules concerned by the chain and `alltargets` is a hashtable that contains for each chain the list of rules whose the chain is the target.

```
for each chain c:
  targets = alltargets[c]
  sources  = allsources[c]
  for each rule_head in targets:
    for each rule_tail in sources:
      result = rule_head.merge(rule_tail)
      if result is a valid rule:
        rules.add(rule_head.index, result)
    rules.remove(rule_head)
    allsources[rule_head.fromchain].remove(rule_head)
  for each rule in sources:
    alltargets[rule.target].remove(rule)
```

**Figure 3.5 –** Algorithm uses by the parser to remove the user-defined chains

This algorithm calls the `merge` function. It corresponds to a method of the class `Rule`. This method is explained in the next section about the parsing process in the rules.

At the end of algorithm, all the rules should not be concerned by any chain and should have as a target ACCEPT or DROP. When all the merging has been done, the rule list is scanned. If a rule is still concerned by a chain or has a chain as target, it must be dropped.

### 3.3.2 Rule parsing

This section is about rule parsing into the `Rule` class and into the classes that extend it.

#### Parsing of a rule

When it is asked to a rule to parse its string representation, it begins by splitting this string to obtain a substring for each match. All the matches start with a «`--matchName`» or «`-matchName`» and can be preceded by «`!`» if they are inverted. When a match is recognised, a method specific to it is called to parse the arguments if any. The following item explains how to deal with these arguments.

#### Supported fields

Here is a list of the fields that are supported by the current implementation and the way they are handled. For most of them, it is possible to invert them with a «`!`» before the entire match or before the value.

**Target** When the target is found, a verification is made on the value before storing it. If the target is LOG, the rule does not give any decision on the packet and so the rule is dropped. On the other hand, if the target is REJECT, it means that the packets will be dropped and that an error message will be sent to the source. So, REJECT can be replaced by a simple DROP. The other values are ACCEPT, DROP or an user-chain; these values are stored.

**Protocol** The protocol stored is the string name of the protocol in lower case.

**Source/Destination IP** The argument can be either a single host IP address or an IP address with a mask.

**Input/output interface** No problem, nothing to check.

**Source/Destination port** In accordance with the protocol, the port or port range is added in the suitable element group (TCP or UDP).

**TCP flags** In *iptables*, all the combinations of the flags SYN, ACK, FIN, RST, URG and PSH may appear. However, only two configurations will be accepted for the parsing in our language: SYN=1, RST=0, ACK=0 and SYN=0, RST=1, ACK=1. The other ones cannot be converted in the language but fortunately they are rarely used. The two recognised combinations can be interpreted as *established* for the first one and *!established* for the other one.

**ICMP type** The value of this match is either only the type or the type and the code of the ICMP messages concerned.

**AH/ESP SPI** The value can either be a unique value or a range. The element has to be added in the AH or ESP group according to the protocol value.

**Conntrack state** The conntrack state values are those of the state machine introduced in 2.6.2. Any combination of the states NEW, ESTABLISHED, RELATED and INVALID may appear but only the values NEW and ESTABLISHED are considered. Four cases can be differentiated:

**«NEW» or «!ESTABLISHED»** are translated to *!established* in the language.

**«!NEW» or «ESTABLISHED»** are translated to *established* in the language.

**«NEW, ESTABLISHED»** is not translated since there is no restriction, all the packets match it.

**«!NEW, !ESTABLISHED»** cannot be translated, no packet will match it. The rule must be dropped.

**Source/Destination IP range** IP Ranges are sets of IP addresses between two values. The problem is that, in the language, a source or destination address can only be defined by what is called one IP address (i.e. a host address and a CIDR mask). An IP range corresponds to a list of such IP addresses and then, cannot be expressed in one rule with conjunctions. So the solution is to extract a list of IP addresses from this range (for instance $1.0.0.0 - 1.0.1.2 = 1.0.0.0/24 \,\&\, 1.0.1.0/31 \,\&\, 1.0.1.2/32$) and to duplicate the rule for each of these addresses.

**Source/Destination ports** This match does not only take as value a port or a port range but also accepts port or range lists. For instance, the following value is authorized: «$3-6, 8, 9-10$». Like in the previous item, the rule has to be duplicated for each of the port or port range.

**Packet type** This match has as value either *unicast* or *multicast* or *broadcast* and match these kinds of packet destinations. Its translation has been simplified. When *multicast* is given, addresses of public multicast groups are matched (224.0.0.0-239.255.255.255), when *broadcast* is given the overall broadcast address is matched (255.255.255.255) and all the other addresses correspond to *unicast*. It is clear that this translation is not really exact but no information has been found on how this match has been implemented. In a further version, this should be improved with an analysis of the interface list for broadcast addresses. However, this approximation is not severe since this match is seldom used.

**IP fragment** This field is easy to convert since it has no argument. It matches the second, third and the following fragments of fragmented IP packets. This is used to drop these fragments since they do not contain UDP or TCP headers and so cannot be easily associated with a flow.

**Limit** This match limits the number of packets accepted per time-unit. This can be used to avoid DoS attacks or to log a sample of the packets. The objective of DFA is not to perform a dynamic analysis but a static one. Then these matches are assumed to match all the packets that pass through the rule. These matches are therefore ignored.

**Physdev** This match is used in bridges to filter on the input or output ports. Since these ports are the equivalent of interfaces of classic host, they are translated as interfaces in the intermediate language. If a value ends with a «+», it matches all the interfaces that begin with the value before the plus. It has not been implemented and so it is not supported. The substitution of it by the related interfaces has to be done by hand.

### Merging two rules

The algorithm to make the chains disappear uses a method to merge two rules. This method is implemented in the *iptables* implementation of `Rule`. The merging of two rules is done by merging each field of the first rule with similar fields of the second one and by conserving all the fields that do not exist in both rules. Merging two values of a field is an AND operation. The way to do it will be different among the four types of data described in section 1.3.4 (IP address, range, unique value and no value). The value often comes with an `inverted` value. If `inverted` is true, the field has to be different from this value and is called *negative* is the next algorithms. The term *positive* relates to the fields that are not inverted.

For unique value fields, 5 cases can be considered:

**The two fields have the same value and are both either positive or negative.**
One of the two field values is kept with its `inverted` value.
Example: $protocol = tcp \ \& \ protocol = tcp \rightarrow protocol = tcp$.

**The two fields have the same value but one is positive and the other negative.**
The two fields are not compatible, no packet will match this and the rules cannot be merged.
Example: $protocol = tcp \ \& \ protocol \neq tcp \rightarrow \emptyset$ : merging of the rules stopped.

**The two fields have different values and are both positive.**
The rules cannot be merged.
Example: $protocol = tcp \ \& \ protocol = udp \rightarrow \emptyset$ : merging of the rules stopped.

**The two fields have a different value and are both negative.**
The two fields are kept.
Example: $protocol \neq tcp \ \& \ protocol \neq udp$ cannot be simplified.

**The two fields have different values and one is positive and the other negative.**
The positive field is kept, the other is ignored.
Example : $protocol = tcp \ \& \ protocol \neq udp \rightarrow protocol = tcp$.

Now, let us deal with the ranges. A range can always be viewed as a set of values. Consider that the sets of values of the two fields to merge are called $A$ and $B$. To do the conjunction of

the two fields is equivalent to do the intersection of the sets ($A \cap B$). Here is an example: the first field is defined by `10:100 & !20:30` and the second one by `25:130 & !70:90`. The result of the conjunction is `31:100 & !70:90`. (Note that such discontinuous range could also be represented by a disjunction (`31:69 | 91:100`), but if it is done the rule has to be splitted into two rules with ranges `31:69` and `91:100`.) The general algorithm used to merge two ranges into one is the following. We suppose that $Z$ is the final value. At the beginning, the whole range is selected ($Z = 0 - 65535$). For each not inverted range $X$, the range $Z$ is restricted to its value ($Z = Z \cap X$). For each inverted range $Y$, a part of the range $Z$ is removed ($Z = Z \setminus Y$). When all the ranges have been added, the resulting range ($Z$) can always be represented with a not inverted range and eventually some inverted ones.

For IP addresses (host IP + mask length), the algorithm used is different. Using the properties of IP addresses seen in 1.3.4, it is accepted that all the IP addresses sets can be represented by one not inverted IP address (the *positive*) and zero or more inverted ones (the *negatives*). At the beginning of the algorithm, we suppose that we have an empty *negatives* set and *positive* equals to the global IP address 0.0.0.0/0. When a not inverted address is added, three different situations may occur. If it includes the current *positive*, nothing changes. If it is disjoint of it, the merging of the rules can be aborted. Finally, if it is included in the current *positive*, it replaces the *positive*. When an inverted address is added, several situations may also occur. If it includes the *positive* one, the rules cannot be merged. If they are disjoint, the inverted address can be ignored. In the other cases, the address has to be added into the *negatives* list. Of course, there are a few igored details in this presentation of the algorithm in order to make it simpler to understand.

As you can see, the merging algorithms are not so easy to implement. Unfortunately, this kind of difficulties involves more risk of mistakes. For this reason, the implementation of this part has been thoroughly tested. The different test sets used are listed in Appendix A.2.

## 3.4 Limitations

Our implementation of the parser works with most unmodified *iptables* configuration files. Nevertheless, some hypotheses are still made on the input configuration whatever the imported language was. Some of these hypotheses has already been brought up before but are reminded because they need small modifications of the input file supplied.

- All the rules of the filtering table that refer to NATed addresses have to be removed. As explained previously, it has been chosen not to consider the address translations. If such rules remain, local addresses could be considered to be located on the global network (on the Internet) and it can lead up to false anomalies.
- For the same reasons, all the rules referring to the loopback interface or to the 127.0.0.1 address have to be removed.
- It must be possible to represent the network structure of the firewall. Actually some firewall cannot be represented with the interface list used in this implementation. An example is a bridge that filters two flows of data that are entirely disjoint. There is no only one but two default routes and this cannot be expressed with our interface lists.
- As explained in the supported fields, the *physdev* values that end with a «+» in *iptables* configuration are not supported. The substitution of the value by the entire names of interfaces has to be done by hand.

- Finally, the input files have to be correctly formulated. If the *iptables* input file is changed by hand, it must have the same shape as an `iptables-save` file. The interface list has also to observe the structure defined. No assumption can be made on the result if the input files do not observe their specifications.

Note that if some fields and targets are not recognise by the parser, warning messages will be displayed in the control GUI. It is always a good idea to check this stream at the end of the parsing.

## 3.5 Support of other firewall configuration languages

Since it is one the objective of the work, the parser part of the application can be extended to support more languages. The main modifications to do are the creation of a `Parser` and a `Rule` classes in a specific package for parsing operations of this language. These classes have to extend the generic ones and have also to implement the non-defined methods of the abstract class.

Other classes will require small changes. Here are theses classes:

**dfa.FwLanguage** This class manages the different languages and their string representations. New languages have to be referred in this class to appear in the language combo box into the GUI.

**dfa.parser.SourceFw** This class opens the files and create the instances of the parsing objects. The class has to know the name of the parser class to call for the new language.

The other classes do not care of the input language of the firewall. The same steps as those explained in this chapter on *iptables* have to be applied for other languages. It is obviously a good idea to use the *iptables* implementation as a model to implement similar operations.

## 3.6 Conclusion

This section described how our parser implementation is organised. Its architecture is hierarchical: a firewall class calls a parser that is made up of a list of rule objects. Then, all the XML child elements of a rule are represented as children in Java tree structures. These structures are managed with the *org.jdom* Java package that allows them to be directly exported to XML files.

More time than expected was spent on this part of the implementation. Indeed, as you may have seen, parts of the implementation contain relatively complex algorithms. The difficulty came from the odd cases that have to be supported although they hardly ever occur. The merging of rules also required complex data manipulations. This chapter has not detailed these long and sometimes boring algorithms since the interest for the reader is not very high.

Chapter 4

# Analysis of distributed firewall configurations

All the components required to perform the main objective of this thesis are now available . Given two firewall configuration files, it is possible to analyse which flows are authorized between two networks. This analysis can be split in two parts. First, it is interesting to obtain, for each data flow between two networks, an overview of the decision made by each firewall. These decisions can be examined manually by a network administrator or by a user of the network to know the authorized connections between the two networks. They can also be automatically analysed by a tool to detect potential anomalies. The main ones will be explained in this chapter as well as the algorithms to discover them.

## 4.1 Overview of the firewall decision for each flow

The aim of this section is, from the firewall configuration files, to list the firewall behaviours on each data flow between two networks. Between two hosts and for the same service (i.e. the same TCP ports for instance) there are four possible decisions: (the two hosts will be called X and Y)

1. Are requests from X to Y authorized?
2. Are requests from Y to X authorized?
3. Are responses from Y to X authorized?
4. Are responses from X to Y authorized?

If everything is coherent, the second item involves the third one and the first one involves the fourth one. Indeed, if requests are authorized, it means that new connections and following packets are accepted. So there is no reason for response packets in the same direction to be rejected.

To answer these four questions, a simple approach is to read linearly the firewall configuration files and to keep the rules matched by the packets of the flow concerned. Once it is done for each firewall, several lists are obtained containing the rules determining the policy for this flow. Then, these lists can easily be interpreted by a human or by a computer.

A good way to have a global overview of the policies applied for all the possible flows is to use a policy tree. The following sections explain what is a policy tree and how to build it efficiently.

### 4.1.1 What is a policy tree

The initial idea of using a policy tree comes from [Al S 03, Al S 04a]. The tree they use is similar to the one decribed in this chapter. Nevertheless, how it is used and built is not exactly the same. The differences are explained in more details at the end of the description of our policy tree.

A policy tree is a tree that represents all the possible flows covered in one or more firewalls. For the analysis needed in this work, it will cover the filtered flows between two firewalls. In the leaves we store the information about the flow. In our case, the leaves will store the rules associated with the concerned flow. Such a tree is represented in Figure 4.1. At each level, the policy tree evaluates a new field. The order chosen for the different fields is arbitrary, however the choice made seems appropriate. The order is the following: IP address of A, IP address of B, protocol and ports of A and B if necessary (under the dashed line). Each branch of the tree corresponds to a specific value or a range of values; a star (*) corresponds to all the other values that are not represented in the branches. Each node is in charge of a new field and has one or more children. Finally, the leaves contain lists of the rules corresponding to the flow represented by the path from the root to the leaf.

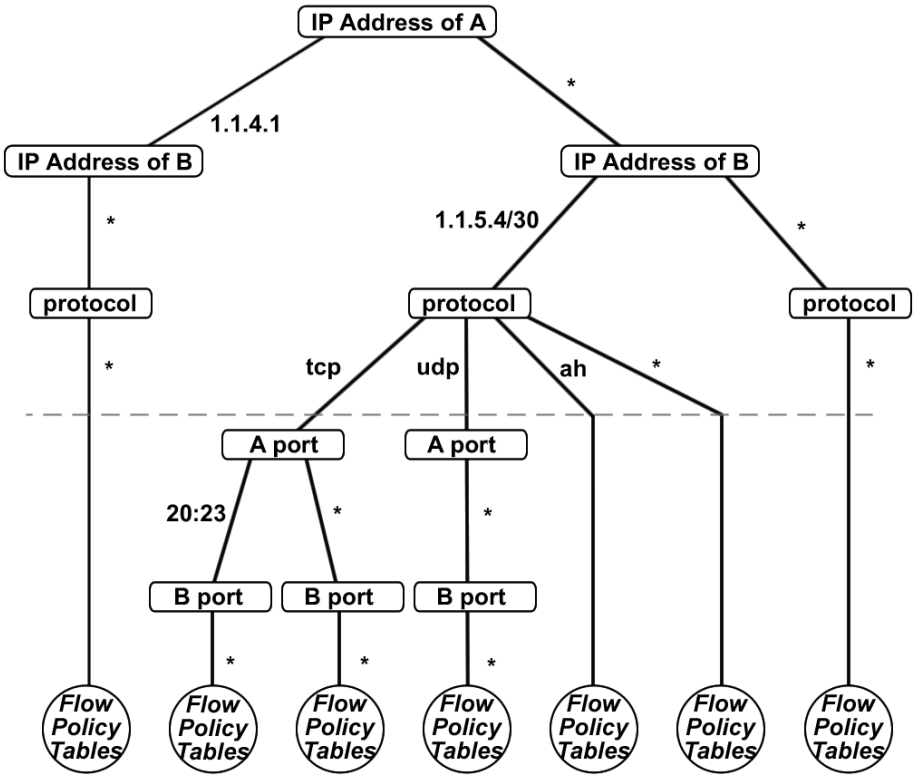



**Figure 4.1 –** An example of policy tree

In the policy tree used by our application, the leaves contain four lists called flow policy tables. Why four? First, because two firewall configurations are analysed and lists are filled while reading each configuration file. It is not a good idea to blend the two flow policy tables since they provide

the information of which table blocks the packets. Moreover, this kind of merging is not as simple as it can seem. Secondly, each of these two lists is divided into one relating to the flow from A to B and another from B to A. Once more, the aim of this division is to make clearer the output and so to simplify the interpretation of the results.

Why not to add nodes for other fields? Some other fields like IP properties may appear in rules and indeed could be added to the tree structure like the other ones. We choose to not add them because that these fields are not very important for packet filtering and do not often appear. If added, these fields would increase the complexity of the tree, its construction and its interpretation. For instance, IP fragmentation is not something differencing a flow from another and so, it is not a good idea to discriminate the nodes based on this property.

In the current version of our implementation, we only take into account the properties of the TCP and UDP protocols. This is because there are fewer ICMP, ESP and AH rules than TCP or UDP ones. Moreover, these rules can be quite easily interpreted even if they are mixed in the same table. For ICMP messages, it is also simpler not to have the rules divided into several tables. Otherwise the ICMP request and response messages could be scattered in different tables.

#### Comparison between our tree and the tree defined in [Al S 03]

The structure of the policy tree described by the authors of [Al S 03] and [Al S 04a] does not differ a lot from ours. The rules inserted in their tree are 6-uples made of the following fields: *action*, *protocol*, *src_ip*, *dst_ip*, *src_port*, *dst_port*. The protocols supported are only TCP and UDP. ICMP, AH, ESP and others are not recognized. IP addresses and ports cannot be made of a conjunction of several values and cannot be inverted.

One major difference between the trees is that theirs does not support any flow information. This is an important simplification since a large proportion of the current firewall configurations uses flow matches.

The field order in our tree is different from theirs: *protocol*, *src_ip*, *src_port*, *dst_ip*, *dst_port*, *action*. It should also be noticed that they use *src_ip* and *dst_ip* on the contrary of our policy tree where *A_ip* and *B_ip* are used. If the packet flows are taken into account, our solution seems to be more relevant.

Finally, their leaf nodes only contain one rule list. There is a single list where are stored the rules disregarding the direction and the firewalls from which the rules come. Moreover, there is only one list for the direction since for their tree structure (with *src_ip* and *dst_ip*), it is sufficient.

### 4.1.2 Construction of a policy tree

The analyser part of the application was made as an independent part from the previous ones. The control GUI takes as input the two firewall configurations (firewall A and firewall B) and generates a policy tree that can be browsed. When a leaf of the tree is selected, the rule lists referring to it are displayed.

Note that it is supposed that the input configurations are well-formed. It means that the input files observe the XML structure defined in the previous chapters. But the hypothesis has also been made that there is no absurdity in the rules defined. For instance, a source or a destination address cannot be defined by two not inverted addresses, a port range cannot be defined as «$100 - 200$ & $!300 - 400$»,... Of course, the files generated by the parser part of the application observe

these rules. This hypothesis is important since some part of the implementation has been simplified thanks to it.

### The Java structure

The implementation of the policy tree is a simple Java tree with a root node that maintains a list of child nodes that also have children and so on. At each level, the nodes are from a different type, the level one contains `IpANode`'s, the level two `IpBNode`'s and the level three `ProtocolNode`'s. The next levels depend on the value of the protocol node. A `ProtocolNode` element with TCP or UDP value has `APortNode`'s children that have `BPortNode`'s as children. The other protocol nodes and the `BPortNode` have `FlowPolicyTables` nodes as leaves. This class records all the rules related to the branch.

To fill the tree, first, the rules are added to the root element of the tree. This element performs several operations:

1. It analyses which IP addresses of A this rule corresponds to. This involves the analysis of the source address, destination address, input interface and output interface fields.
2. It splits if necessary its own children to have at least one node corresponding to each IP address matched in the rule.
3. For each child included in the set of IP addresses concerned by this rule, the root node adds the rule to it.

When the root node has added a rule to one of its children, the child node (an `IpANode` element) performs the same type of analysis for B IP addresses and transmits the rule to some of its children and so on. When a leaf node is reached, the rule is added to a suitable list.

The details of the Java implementation of this part will not be given here but a global idea of the more complex algorithms used are explained in the next sections.

### Interpretation of interface fields into host address information

In order to insert a rule in the tree, its input and output interface fields have to be translated into IP addresses. For illustration, imagine in the network of Figure 4.2 that the firewall B contains a rule `input-if=vlan0 => ACCEPT`. How to insert this rule into the policy tree? By intuition, it seems that an equivalent rule with IP addresses should be `src-ip=1.5.100.* => ACCEPT`. But it is not always so easy. The interface `eth0` of firewall B is associated with network 0.0.0.0/0 but a rule `input-if=eth0 => ACCEPT` cannot be translated into `src-ip=0.0.0.0/0 => ACCEPT`: this solution would match more packets than expected.

So, a more precise algorithm is needed to identify the addresses concerned by a rule when an interface field is mentioned. As previously explained, each interface is defined by the list of networks it leads to. The interface used by a packet to reach a destination is the more specific route for it. Some notations are introduced to make the algorithm clearer:

- $N^A$ is the set of the interfaces of the firewall A.
- $N_i^A$ is the interface of A with index $i$.

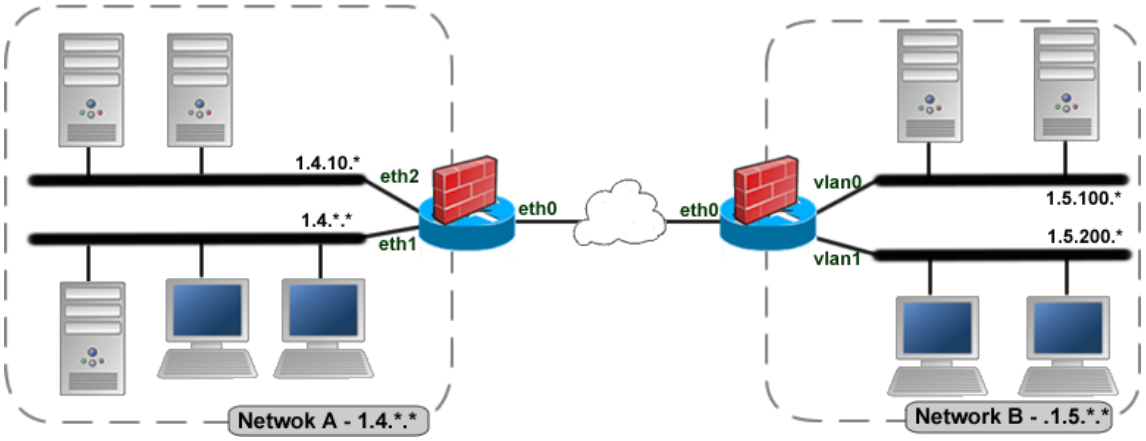



**Figure 4.2 –** An example of a network where interfaces may cause some problems

- $N_{ij}^A$ is j-th network associated with $N_i^A$.
- $N_B^A$ is the interface of A used by packets to reach B. The way to determine it is described in Figure 4.3.

```
Baddr = one of the host addresses of the firewall B
cidr = 0
for each interface N_i^A:
  for each network N_ij^A of the interface:
    if Baddr ∈ N_ij^A and N_ij^A.cidr ≥ cidr :
      N_B^A = N_i^A
```

**Figure 4.3 –** Algorithm used to determine the value of $N_B^A$

So, the algorithm used to translate interface information into IP addresses for firewall A is shown in Figure 4.4. The algorithm is the same for the other firewall, A and B have just to be inverted.

The resulting IP addresses are the ones contained in the $IpA$ set. The set $direction$ is also important. It contains the direction concerned by this rule. If $direction$ is empty at the end of the algorithm, the rule can be dropped (not inserted). This direction is important in the policy tree to select to which tables of the leaf the rule must be appended.

Here is additional explanation on some parts of the previous algorithm:

**(A)** At first sight, this operation seems difficult to understand. Actually, the explanation is quite easy but the implementation does need more attention. When an input interface is given and is not inverted, it is obvious that the IP network addresses behind this interface have to be accepted as source addresses. But if the algorithm is stopped here, there is a problem when an interface network is included in another. This could be illustrated with Figure 4.2: if a rule is considering the input interface `eth1` of A and if this match is replaced by the input address 1.4.*.*, the packets from `eth2` will be also taken into account. To solve the problem, the interface is not only replaced by the addresses of its networks but is replaced by

```
direction = "A → B" ∪ "B → A"
IpA = whole range of IP addresses
for each input interface information I_k:
  if I_k == N_B^A:
    if I_k not inverted:
      direction = direction ∩ "B → A" /* restrict the direction */
    else:
      direction = direction ∩ "A → B"
  else:
    if I_k not inverted:
      direction = direction ∩ "A → B"
      IpA = IpA ∩ ⋃_j (N_kj^A \ ⋃_{m,n,m≠k,N_mn^A ∈ N_kj^A} N_mn^A) /* explained further in (A) */
    else
      duplication of (direction, ipA) in 2 cases:
        (direction1, IpA1) and (direction2, IpA2) /* explained in (B) */
      direction1 = direction1 ∩ "A → B"
      IpA1 = IpA1 ∩ ⋃_{m,n,m≠k} (N_mn^A \ ⋃_j N_kj^A) /* explained in (C) */
      IpA2 = IpA
      direction2 = direction2 ∩ "B → A"
for each output interface information I_k:
  if I_k == N_B^A:
    if I_k not inverted:
      direction = direction ∩ "A → B"
    else:
      direction = direction ∩ "B → A"
  else:
    if I_k not inverted:
      direction = direction ∩ "B → A"
      IpA = IpA ∩ ⋃_j (N_kj^A \ ⋃_{m,n,m≠k,N_mn^A ∈ N_kj^A} N_mn^A)
    else
      duplication of (direction, ipA) into 2 cases:
        (direction1, IpA1) and (direction2, IpA2)
      direction1 = direction1 ∩ "B → A"
      IpA1 = IpA1 ∩ ⋃_{m,n,m≠k} (N_mn^A \ ⋃_j N_kj^A)
      direction2 = direction2 ∩ "A → B"
      IpA2 = IpA
```

**Figure 4.4 –** Algorithm used to translate interfaces into IP addresses

the addresses of its networks minus the networks of the other interfaces that are included in them. To give an example, the interface `eth1` of Figure 4.2 should be replaced by network 1.4.*.* minus 1.4.10.*.

**(B)** This duplication is needed because the rule may match two opposite cases. For instance suppose a match «`input-if!=eth2`» on Figure 4.2. It means that either the packets from network B to network A are authorized or the packets from A to B but only those that do not come from 1.4.10.*. The two cases have to be added separately in the tree.

**(C)** The reasoning about this operation is the same than in (A) except that the sets are inverted.

### Interpretation of the source and destination IP addresses

The interpretation of the source and destination IP addresses is easier than the interfaces. The work here is limited to determine if the source or destination address has to be associated with A or B; remind that the policy tree contains «A address» and «B address» nodes. Table 4.1 summarises the translation from IP source or destination to IP of A or B.

| Firewall | Rule encountered | A address | B Address | Direction |
|---|---|---|---|---|
| A | sce-IP=a.b.c.d | a.b.c.d | * | A→B |
| | | * | a.b.c.d | B→A |
| | dst-IP=a.b.c.d | a.b.c.d | * | B→A |
| | | * | a.b.c.d | A→B |
| B | src-IP=a.b.c.d | a.b.c.d | * | B→A |
| | | * | a.b.c.d | A→B |
| | dst-IP=a.b.c.d | a.b.c.d | * | A→B |
| | | * | a.b.c.d | B→A |

**Table 4.1 –** Translation of the source/destination addresses into A/B addresses

In this table, the star (*) represents all the IP addresses. It may be noticed that all the rules are interpreted by two rules, one in each direction. This is not needed at all, in fact, each rule should be interpreted by zero or one rule because a.b.c.d cannot be at the same time behind the firewall A (seen from B) and behind the firewall B (seen from A). One or the two rules could be ignored thus. However, the choice was made not to make this simplification here in the algorithm; It will be rather done by pruning the policy tree later. If the match is inverted, the a.b.c.d address match is also inverted in the policy tree insertion algorithm.

### Building the tree: adding address nodes

Once the IP addresses of A and B have been found for a rule, the rule has to be inserted in the policy tree. As explained in a previous section, the root node has `IpANode` children and each of these nodes have `IpBNode` children. Before the first rule is added, there is only one A child that has one B child; each of these two nodes match all the IP addresses. So these nodes have to be split when rules are added. The algorithm to do it is described in this section.

First, whatever the number of the fields relating to addresses, their type and whether they are inverted or not, the IP address set matched by a rule can always be represented by one positive (not inverted) IP address and several negative (inverted) ones. The algorithm presented in Figure 4.5 allows to add in a root node a rule that match as a set of addresses symbolised by $A$. The set $A$ is represented by a positive network and several negatives ones. For instance, it can represent the addition of a rule that matches the IP 1.4.*.* except the IPs 1.4.10.* and 1.4.0.1.

Here are a few notations:

- $A^+$ is the positive network.
- $A_i^-$ is the i-th negative network ($1 \leq i \leq n^-$).
- $T_j$ is the network associated with the j-th child of the node where the rule has to be added (the root node).
- $A^+.needsplit/A_i^-.needsplit$ is false if a node with this address already exists, true otherwise. It is initialised to true.
- $A^+.fatherNode/A_i^-.fatherNode$ contains the reference to the child with the smallest network (greatest CIDR) that includes this network.

#### Adding protocol and port nodes

The three previous sections examined the IP addresses in the policy tree. But once this has been done, the lower levels have to be filled: protocol and, for UDP and TCP, A and B ports.

For protocol, the extraction of the information from the rules is much easier than for IP addresses. All the protocol information of a rule is contained in the `protocol` marker, if it does not exist, the protocol value is *any*. It can be supposed that the more complex configuration that can occur is the one that rejects several protocols, for instance «`prot!=udp, prot!=tcp`». The node insertion of a rule is done first by adding a child for each protocol used and that does not exist yet (by duplication of the *any* child node). Then, the rule is added to each node matched by the protocol values of the rule.

The same operation for ports is more complicated. The extraction of the information from rules is quite simple: it is contained into the `srcport` and `dstport` elements of the `tcp` or `udp` group. But how are `srcport` and `dstport` translated in A and B ports? Source ports are converted into A port if the direction is $A \rightarrow B$ and into B port if it is $B \rightarrow A$. The opposite is done to translate the destination ports. If no direction is defined, the rule is add to each direction. Concerning the insertion of the information in the tree, each port node has a port range associated with it. These ranges have to be split if necessary when new values are inserted. For example, if the existing nodes cover the ports 0-1024 and 1025-65535 and a rule concerning ports 500-510 is inserted, the first range has to be split into three sub-ranges. If the rule inserted is 1000-2000 instead, the two ranges have to be split into two sub-ranges. The detailed algorithm will not be listed here because it is quite long and relatively similar to the previous ones.

### 4.1.3 Improving the efficiency of the policy tree

The problem with the general structure of the policy tree implemented was especially that the resulting tree was too large. It was insignificant for small firewall configurations used for tests but was problematic in the real configurations used for validation. For example, the insertion in the tree of a 400 rules firewall configuration and a 1175 rules one requires several GBytes of memory to store the tree. In fact, in these examples, more than ten millions of nodes were used. The whole tree needs to be loaded in memory to be explored easily and to permit reading the rules associated with each leaf. The following items give interesting improvements to the tree in order to reduce the number of nodes needed in memory at the same time.

```
addRule = empty_list // list of the nodes concerned by the current rule

for each T_j :
  if T_j == A+ :
    A+.needsplit = false
    addRule.add(T_j)
    for each A_i- :
      if A_i-.needsplit and A_i-.fatherNode.cidr ≤ T_j.cidr :
        A_i-.fatherNode = T_j
        A_i-.fatherNode.cidr = T_j.cidr

  if T_j ⊃ A+ :
    if A+.needsplit and A+.fatherNode.cidr ≤ T_j.cidr :
      A+.fatherNode = T_j
      A+.fatherNode.cidr = T_j.cidr
    for each A_i- :
      if A_i-.needsplit and A_i-.fatherNode.cidr ≤ T_j.cidr :
        A_i-.fatherNode = T_j
        A_i-.fatherNode.cidr = T_j.cidr

  if T_j ⊂ A+
    addRuleForThisNode = true
    for each A_i- :
      if T_j == A_i- :
        A_i-.needsplit = false
        addRuleForThisNode = false
      if T_j ⊃ A_i- :
        if A_i-.needsplit and A_i-.fatherNode.cidr ≤ T_j.cidr
          A_i-.fatherNode = T_j
          A_i-.fatherNode.cidr = T_j.cidr
      if T_j ⊂ A_i- :
        addRuleForThisNode = false
    if addRuleForThisNode :
      addRule.add(T_j)

for each A_i- with A_i-.needsplit == true :
  newNode = duplicate(A_i-.fatherNode)
  newNode.changeValue( A_i )
  addNewChild(newNode)
  addRule.add(A_i-.fatherNode)

if A+.needsplit == true :
  newNode = duplicate(A+.needsplit)
  newNode.changeValue( A_i )
  addNewChild(newNode)
  addRule.add(newNode)

for each N_i in addRule :
  N_i.addCurrentRule
```

**Figure 4.5 –** Algorithm used to add IP addresses in the tree

#### Do not add rules in branch that cannot be matched

As explained in the section about the insertion of source and destination IP in the tree, all the addresses are added in the tree disregarding whether these addresses really belong to the network. If a rule is related to an address that is not behind the firewall A seen by firewall B, the rule has not to be added in the *A address* nodes with this value. In the example of networks A 1.0.0.0/8 and B 2.0.0.0/8, the only *A address* nodes that can be created are the ones with an address included in the network A (1.0.0.0/8).

To apply this improvement in the tree, the following algorithm has been followed. At the beginning of the tree building, *A address* nodes are created for each of the networks associated with the interface of firewall A to B (often 0.0.0.0/0). These nodes are defined as locked. It means that these nodes exist but do not add new rule and create children anymore. Then, all the networks of the other interfaces of A that are included in these locked networks are also inserted as *A address* nodes but they are unlocked.

The same algorithm can be applied for B nodes. It importantly reduces the number of nodes in the tree.

#### Do not add rule in nodes that already have a global policy

This improvement comes from an observation: In each of the four tables of the leaves, a lot of rules are stored despite that the policy is definitely fixed after a few rules. For illustration, if the rule `src-ip=1.1.0.0/16 => ACCEPT` has already been added in a leaf, the rule `src-ip=1.0.0.0/8 => DENY` does not need to be added as well as any new rule with a source address included in 1.0.0.0/16. In the other hand, if the rule `src-ip=1.1.0.0/16 & established => ACCEPT` has been added, other rules are welcomed to complete the policy. This problem can easily be solved with a few variables that save whether a table is entirely defined or not. A rule decides of the final policy of a table if the fields of the rule are only among input and output interfaces, source and destination addresses, protocol and source and destination ports (the fields used for the construction of the policy tree). If another field is defined, the established property for example, nothing can be concluded from this rule in the table. So, a table of a leaf is locked when it becomes entirely defined. When a table is locked, no rule is accepted anymore.

Another observation made was that some nodes are uselessly split for the same reason. For example, a rule `src-ip=1.1.1.1 & dst-ip=2.1.1.1 & protocol=tcp => ACCEPT` is inserted first and then `src-ip=1.1.1.1 & dst-ip=2.1.1.1 & protocol=tcp & src-port=15 => ACCEPT` is also inserted. The second rule will involve the spitting of the port node «*» in 3 three nodes («0-14», «15» and «16-65535»). The splitting is absolutely useless because all the tables of the child nodes are already locked. To avoid this, when a leaf locks one of its tables, it warns its parent. When the parent is warned, it checks all its children and, if they have all locked one of a table, the parent put also a lock on the table. Each table corresponds to a couple *(firewall, direction)*; a node that has locked a table does not add packets corresponding to this couple anymore.

#### Merge the `FlowPolicyTable` nodes with their parents

The `FlowPolicyTable` nodes were the leaves of the policy tree. The leaves in such tree are the more numerous nodes; there are several millions of such nodes in trees of thousands of rules.

By conception (see before), the parents of these leaves have always only one child. So, a way to reduce the total number of nodes is to merge the leaves with their parents. So the tables are now contained in *B port* nodes or in *protocol* ones when the protocol value is different from TCP and UDP.

#### Explore one branch at time

Once these enhancements have been implemented, the number of nodes was widely reduced but not enough. The nodes needed for a problem with 400 rules for A and 1175 for B was more than three millions and the memory needed around 1.3 GByte. To be able to run the application on a standard computer, a last improvement has been made. When the analysis is launched, all the rules are handled but no node is created beneath the first level. Then, the user can choose the addresses of A he wants to explore. The second analysis is limited to the second level, A address and B address nodes are explored and the choice is given to the user to select the B network. Then, the branch of the tree selected is entirely explored and can be freely browsed. The choice is always left to the user to explore all the branches but it works only for small configurations or on powerful computers.

## 4.2 Anomaly detection

The preceding tool is interesting to analyse two simple firewalls or to have a good idea on the policy applied to a specific flow but it does not provide a good support to find easily real errors. That is why it could be useful to have a tool that points the abnormal rule configurations to help the administrator to detect the actual errors.

### 4.2.1 Anomaly classification

A lot of remarks could be made on distributed firewalls configurations. However, a lot of those are not real anomalies but are due to the complexity of the configurations. Some rules could also look strange only because they have been written with a special care of efficiency or of human readability.

The article [Al S 04a] outlines four anomalies that can be discovered in distributed firewalls[1]. Here is a short summary of them. It supposes that the upstream firewall is the one nearer to the source.

**Shadowing:** It occurs when the upstream firewall blocks packets accepted by the downstream one.

**Spuriousness:** It occurs when the upstream firewall accepts packets dropped by the downstream one.

**Redundancy:** It occurs when the upstream firewall blocks packets already blocked by the downstream one.

[1] Remember that this article does not take the flows into account.

**Correlation:** It occurs when there are two correlated rules in the upstream and downstream firewalls. Two rules are correlated if some fields of the first rule are subsets of those in the second one and if some fields of the first rule are supersets of those in the second one.

The shadowing and the spuriousness are interesting anomalies to match and they will be explained in further details later. However, the relevance of the redundancy anomaly can be discussed. If it is taken into account, it involves that a blocked flow will always match one of the first three anomalies. But is it an anomaly to have a flow blocked and is it abnormal to have packets blocked in both firewalls of a distributed firewall configuration?

One situation where redundancy could be an anomaly is represented in Figure 4.6. It is a configuration where the network between the two firewalls is trusted. Packets coming from the WAN and passing through the firewall A do not need to be checked again in firewall B. However, it could be considered that this kind of situation is very special and is seldom encountered. Actually, this configuration is a subset of the configurations taken into account. So, all the anomalies found by the application will be relevant in this kind of configuration but some anomalies could be missed.

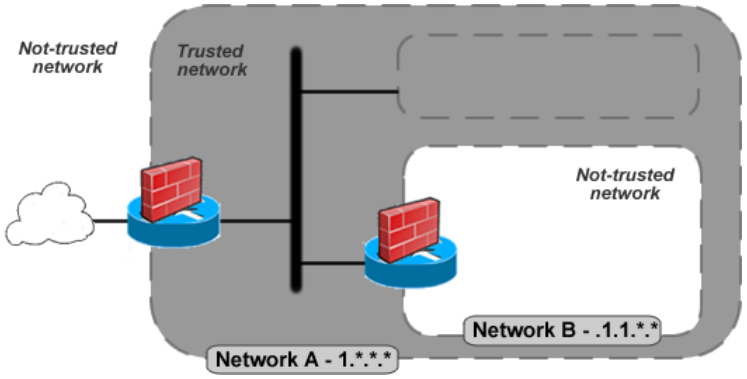



**Figure 4.6 –** A network in which the redundancy anomaly detection could be relevant

Since the redundancy is not an anomaly in classical distributed firewalls, it will not be considered in such way. About the last anomaly listed in [Al S 04a], the correlation is above all a special case of shadowing and spuriousness. For this reason and because correlation is unavoidable in a complex firewall configuration, correlation will not be considered in itself as an anomaly.

In addition to these anomalies, some other ones considering flows could be defined. The first one is «blocked replies». Apart from some special needs, there is no reason to let packets pass in one direction and to drop them in the other one. Another anomaly to consider is about the misuse of the *established* property. There is no point to accept established flows if the flow is not accepted in the other direction. The items below describe more in details each anomaly considered and detected in DFA.

### Upstream-blocked anomaly

The *upstream-blocked* anomaly corresponds to the shadowing anomaly of [Al S 04a]. It occurs when the upstream firewall (nearer to source) blocks packets that are accepted by the downstream firewall. A representation of this kind of anomaly is shown at Figure 4.7.

The security risk with this kind of anomaly is not very important but it reveals incoherence between the two configurations and could permit unexpected spoofing attacks. The anomaly is

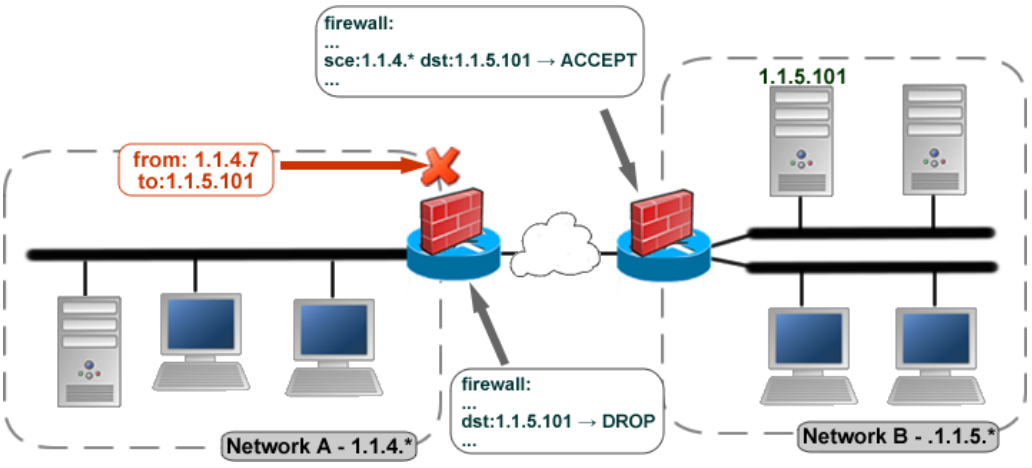



**Figure 4.7 –** A simple example of upstream-blocked anomaly

more critical if, as in the example, the accepting rule is related to the source network that has blocked the flow.

### Downstream-blocked anomaly

This anomaly is equivalent to the spuriousness anomaly of [Al S 04a] and so, occurs when the downstream firewall blocks packets accepted by the upstream one. This anomaly is represented at Figure 4.8.

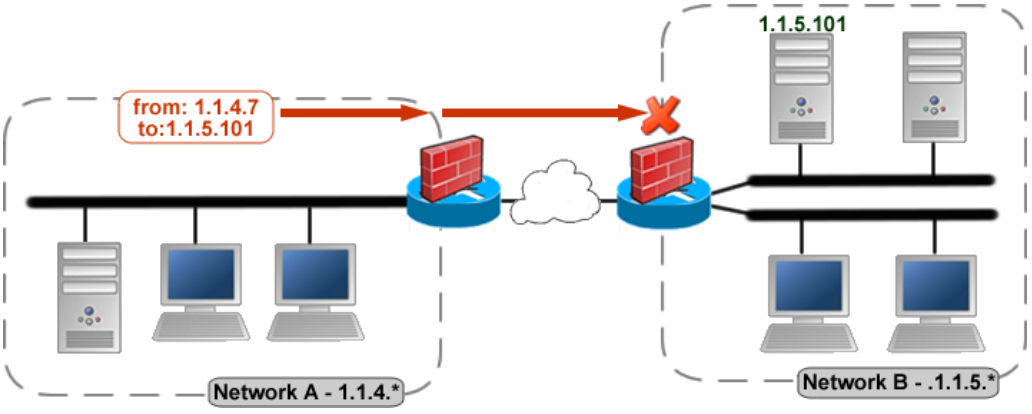



**Figure 4.8 –** A simple example of downstream-blocked anomaly

This anomaly reveals also incoherence between the two firewall configurations. The main problem of this kind of anomaly is that packets pass through the network between the two firewalls for nothing. If the communication is limited or has to be paid, it could be important to drop directly the packets in the first firewall. As in the previous anomaly, it is more critical if the accepting rule concerned precisely the network that rejects the packets.

### No reply anomaly

The *no reply* anomaly occurs when requests are accepted from network A to network B and replies are blocked either in the first or in the second firewall. A representation of this anomaly is shown on Figure 4.9.

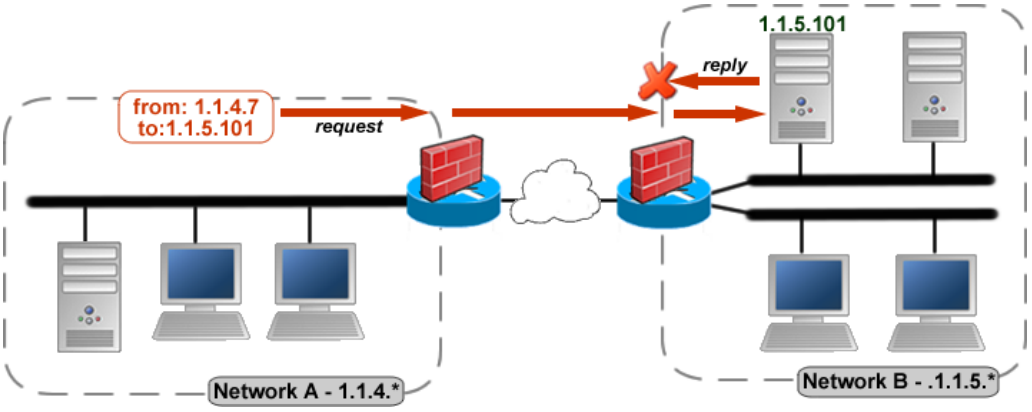



**Figure 4.9 –** A simple example of the *no reply* anomaly

Most of the time, this anomaly reveals a real problem because a very few services works without neither reply nor acknowledgment. If requests are accepted, the packets pass through the network for nothing and possibly create state in the destination machine. If it is pointlessly sent, a better security policy should be to reject the requests too. Note that if this kind of anomaly is found in distributed firewalls, the anomaly can also be found in one of the two firewalls independently of the other one.

### Useless established anomaly

The last anomaly matches the futile uses of the *established* keywork. This anomaly is encountered when the *established* of an accepting rule could be replaced by a *drop* without changing the policy of the distributed firewall. A situation in which this anomaly occurs is represented at Figure 4.10. All the *established* are useless in this figure: they could be replaced by a drop rule without changing anything in the policy between the two firewalls.

Once more, such anomaly does not reveal any severe security risk but points a potential misunderstanding between the administrators of the networks. In the example, it would be possible that the administrator of firewall A believes that the flows initiated by A to B are authorized while they do not.

## 4.2.2 Anomaly searching

The next step of the analysis is now to determine where the defined anomalies occur in the given configurations. To do this, the application will base its analysis on the policy tree previously designed. As a reminder, each node of the policy tree has four tables, one for each firewall and one for each direction. These tables contain a list of rules that correspond to it. The last one of

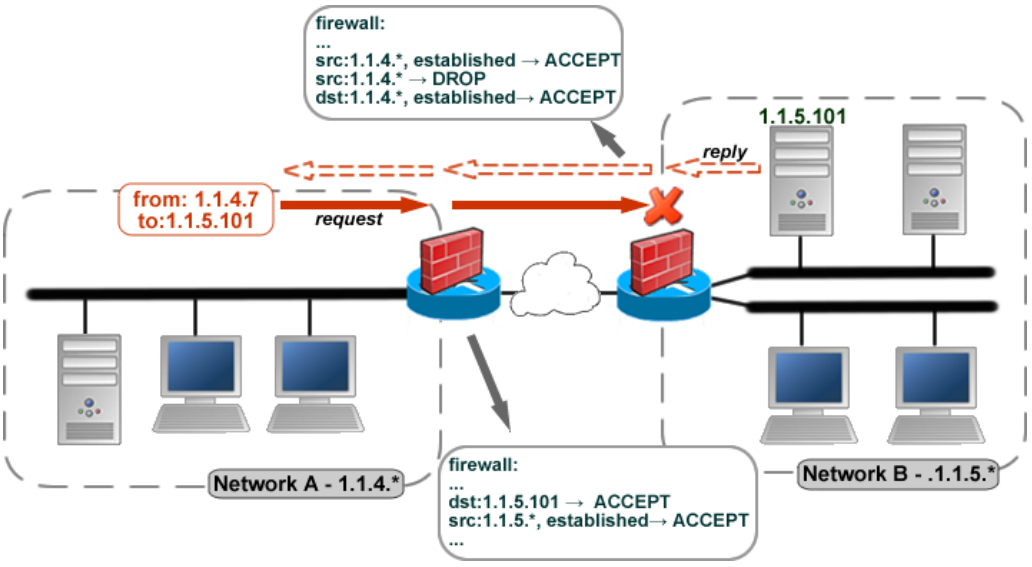



**Figure 4.10 –** An example of the *useless established* anomaly

these rules is always a global rule that matches all the packets concerned by the table[2]. In the leaves concerning the TCP or UDP flows, tables contain a few rules, when there are more than one rule, it is often due to a rule concerning the established flow. In the leaves of the other protocols, more rules can appear. For example, an ICMP leaf may describe the action to apply to each ICMP message before defining the default behaviour.

A great simplification is made at this point. The policy associated with a table will be the policy of the last rule of the table, this is the default policy. However, if the default policy is DROP and there is a rule before that is defined as `established => accept` or if the default policy is ACCEPT and there is a rule before that is defined as `!established => drop`, the policy of the table will be considered as ESTABLISHED. So the policy associated with a table is either ACCEPT, DROP or ESTABLISHED.

The general policy associated with a flow is represented by the four policies. It could be represented by a policy table as Table 4.2. In this table, A|D|E is one of the three policies ACCEPT (A), DROP (D) or ESTABLISHED (E).

| | A | B |
|---|---|---|
| A→B | A\|D\|E | A\|D\|E |
| B→A | A\|D\|E | A\|D\|E |

**Table 4.2 –** Model of a policy table

The policy table corresponding to a flow allows to detect the anomalies. Since the number of different policy tables is finite, the matching between a policy table and the known tables is sufficient to do an efficient anomaly detection.

[2]It could be considered that all the tables always finish with a global rule that matches all the packets of this table. If no global rule exists for a table, it would mean that the hypothesis saying that all the packets match at least one rule would not be respected. Theoretically the hypothesis does not require having a global rule, several rules that cover all the cases should be sufficient. However, in nearly all the possible configurations a global policy exists, translated from the default policy.

Before looking at how to recognise each anomaly with the policy tables, note that the tables represented at Table 4.3 are equivalent by swapping A and B. The four letters in the tables (a, b, c, d) represent one the three policies.

| | A | B |
|---|---|---|
| A→B | a | b |
| B→A | c | d |

| | A | B |
|---|---|---|
| A→B | d | c |
| B→A | b | a |

**Table 4.3 –** Two equivalent policy tables

#### Anomalies detected into configurations

Table 4.11 represents all the different possible configurations and the anomalies associated with them. When `D/E` is indicated, the same anomalies occur whatever the value is `D` or `E`. However, if the value is `E`, the result is the same than `D` and so a *useless established* anomaly occurs.

### 4.2.3 Relevance levels of anomalies

By matching the previous patterns with the policy tables, it is now possible to list all the anomalies of a configuration. If these anomalies are examined, a few real errors will certainly be found and a lot of the anomalies are well known from the administrator and do not cause any problem.

For instance, consider the network represented at Figure 4.8. If the *accept* rule in the firewall A is `dst-ip=1.1.*.* => ACCEPT` and the *drop* rule in B is `dst-ip=1.1.5.101 => DROP`, the anomaly is not very important. It can even be supposed that this anomaly is intentional not to overweight the filtering table of A. Indeed if the table of A has to contain DROP rules for each destination on the Internet that can drop its packets, it would contain millions of rules. On the contrary, if the ACCEPT rule in the firewall A is `dst-ip=1.1.5.101 => ACCEPT` and the DROP rule in B is `dst-ip=1.1.5.* => DROP`, the anomaly is more critical. It can be assumed that the configuration of the firewall has been made without the knowledge of the policy of B. The contrast between these two configurations about the same anomaly shows that there are several degrees in the anomaly relevancy.

For this reason, several levels of anomalies should be defined. The rating of the anomaly should be based on a few arbitrary properties. A good idea could be to compare the mask lengths of the IP addresses used in rules that make the final decision of the dropping or accepting. It could be noticed that the source addresses is the major property to evaluate the relevance of the upstream-blocked anomaly while the destination addresses is for the downstream-blocked one. Unfortunately, by lack of time, it has not been possible to conceive and implement a full relevancy level system for anomalies.

The mechanism implemented allows to detect only some anomalies that can be considered as real problems. These anomalies detected are those of the examples shown above. Four cases are considered:

- There is an *upstream-blocked* anomaly and the destination address of the rule that drops in the upstream firewall is included in a network behind the other firewall.
- There is an *upstream-blocked* anomaly and the source address of the rule that accepts in the downstream firewall is included in a network behind the other firewall.

| | A | B | |
|---|---|---|---|
| A→B | A | D/E | • Downstream-blocked |
| B→A | D/E | D/E | |
| A→B | D/E | A | • Upstream-blocked |
| B→A | D/E | D/E | |
| A→B | A | A | • - (no anomaly) |
| B→A | E | E | |
| A→B | A | A | • No reply anomaly |
| B→A | D/E | D/E | |
| A→B | A | D/E | • Downstream-blocked<br>• Upstream-blocked |
| B→A | A | D/E | |
| A→B | A | A | • No reply anomaly<br>• Upstream-blocked |
| B→A | A | D | |
| A→B | A | A | • Upstream-blocked |
| B→A | A | E | |
| A→B | A | D | • Downstream-blocked<br>• No reply anomaly |
| B→A | A | A | |
| A→B | A | E | • Downstream-blocked |
| B→A | A | A | |
| A→B | D/E | A | • Upstream-blocked (in both directions) |
| B→A | A | D/E | |
| A→B | A | D/E | • Downstream-blocked (in both directions) |
| B→A | D/E | A | |
| A→B | A | A | • - |
| B→A | A | A | |
| A→B | D/E | D/E | • - |
| B→A | D/E | D/E | |

**Figure 4.11 –** All the possible configurations and their anomalies

- There is an *downstream-blocked* anomaly and the destination address of the rule that accepts in the upstream firewall is included in a network behind the other firewall.

- There is an *downstream-blocked* anomaly and the source address of the rule that drops in the downstream firewall is included in a network behind the other firewall.

No real problem is detected on no-reply anomaly since this kind of anomaly involves a problem in one of the firewalls independently from the other. This software was not developped to make verification of single firewalls and so no energy has been spent on detection of this kind of anomaly.

About the implementation, there are not a lot of things to report. The source or destination address set associated with a rule may include inverted and not inverted addresses. In such cases, only the not inverted address is considered since the inverted ones are included in it. The other manipulations has already been explained in other sections.

## 4.3 Conclusion

The policy tree provides a quick overview of the policies applied by one or several firewalls. This tree can however be used for other purposes than distributed firewall analysis. Only some parts of it would have to be changed in order to be used with a single firewall analyser.

The anomalies introduced in this chapter are only the more obvious ones but many others could be found. These anomalies have to be tested on real firewall configurations to measure their relevance. The next chapter performs their evaluation and proposes other ideas.

Chapter 5

# Validation of the application

Now that the tool has been implemented, it has to be tested on real size network configurations. This section discusses the evaluation of the entire application on a large network that contains three firewalls. The three configurations are coming from real networks and can contain some errors. To maintain the privacy of these firewalls, the names and rules used have been modified.

## 5.1 Overview of the test network

The test network is quite similar to the networks viewed as examples throughout this report. It is represented in Figure 5.1. The network is composed of 3 subnetworks, each one protected by a firewall. The network A is a large network that contains a lot of hosts. It is split in several VLANs to restrict the rights between different subnetworks. The network B is a simple network that is not subdivided but contains both servers and normal computers.

The network C is a bit special. It contains two LANs that are separately processed, even in the firewall. If a packet is going from LAN C1 to LAN C2, it has to leave the network C and thus to pass through the firewall two times. This structure permits to handle the two subnets as entirely different networks. Actually, the firewall C is a bridge with only two interfaces. On each interface two VLANs are defined and each VLAN on an interface is exactly linked with one on the other interface. The VLAN number associated with a packet cannot change in the firewall and thus, although a part of the filtering rules is the same, the two flows can be considered as totally separate flows. A description of this kind of configuration is given in [Grid].

In DFA, an interface list has to be defined to generate the XML configuration file. In this list, among others, appears an interface that leads to the default route. In the firewall C, there are two default routes, one for each VLAN. Thus it is impossible for the analyser to determine which interface is supposed to be used to join the Internet. Actually this route depends on the input interface but this kind of consideration is not known by the analyser. To solve this problem, the filtering table of the network C has to be split into two tables, one for each VLAN. To do this, some minor changes have been made in the table without changing anything in the firewall policy.

The firewall configuration files supplied for the three firewalls are naturally *iptables* ones. The `route -n` and `ifconfig` outputs have been provided and are sufficient to obtain the entire network information needed for firewalls A and B. For the firewall C, the bridge, more information

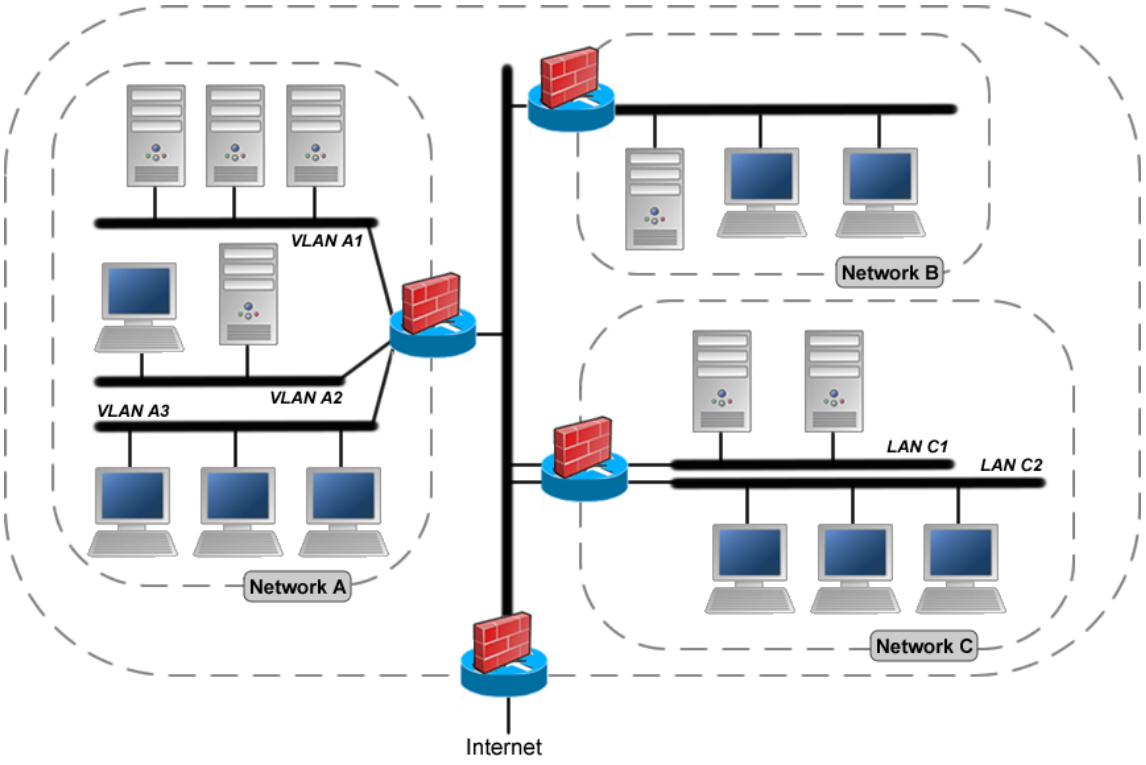



**Figure 5.1 –** A representation of the test network

was required about the addresses associated with each VLAN. The number of rules of A, B and C is respectively 1178, 390 and 638.

## 5.2 Parsing of *iptables* files

No special problem has been encountered during the parsing of the *iptables* configurations. However, the files have to be cleaned before importing them in the application. The main items to check in each table are the following:

- Remove all the rules concerning local addresses that are used for NATing or that can only be accessed from the inside. It comes to remove all the rules relating to networks addresses included in 10.0.0.0/8, 172.16.0.0/12 or 192.168.0.0/16. (37 rules in A, none in B and C)
- Remove the rules about the *loopback* interface (`lo` under Unix) or using the 127.0.0.1 address. (2 rules in A, 0 in B and 1 in C)

To create a filtering table for each VLAN from the firewall configuration C, the only thing to do is to remove the rules that contain references to the other VLAN (`physdev` matches). Such rules frequently have a chain as a target. If only this rule relates to this chain, all the rules starting from the chain will automatically be removed since the chain is never matched. There is however a problem with this splitting, this is the way to deal with the firewall host address. The bridge has only one address and this address is located in the network LAN C1. When the LAN C2 is isolated from the other network, the network address of the firewall is a source of errors. Indeed, a firewall

address inside the network LAN C1 is interpreted as if the LAN C2 was included in C1. At least one firewall address is needed to position the firewall in the network. So to solve the problem, a fake address included in the C2 LAN is associated with the firewall. This address cannot be used in the filtering rules.

Nearly all the rules of filtering tables are well understood. Only the *ctorigdst* match[1] is not recognized, this match appears four times in firewall A. The rules containing this match are dropped. It does not widely affect the policy since the rules concerned did match very specific IP addresses.

The three firewall configurations use user-defined chains. Table 5.1 summarises the effect of the chain merging on the number of rules. The number of log-rules is also mentioned since a slew of such rules can reduce in an important way the number of effective rules. The size of the output of C1 and C2 is smaller than the input for the reason explained earlier. This table permits to have an idea of the number of rules in classical firewalls as well as the number of chains and dropped log-rules.

| Firewall | Size (# IP addr.) | Input rules | User-chains | Log-rules | Output rules |
|---|---|---|---|---|---|
| A | $\sim 640$ | 1139 | 135 | 163 | 7566 |
| B | $\sim 1024$ | 390 | 4 | 0 | 389 |
| C1 | $\sim 64$ | 629 | 12 | 7 | 511 |
| C2 | $\sim 512$ | 598 | 12 | 7 | 359 |

**Table 5.1 –** The number of rules after parsing

## 5.3 Analysis of configurations

### 5.3.1 General overview

The firewall configurations obtained in the preceding section can now be analysed with our application. For example, we will perform the analysis between firewall A and firewall B. Since the two firewalls contain a lot of rules, we will explore only one branch of the policy tree at the same time. Let us look at the flows between the network corresponding to the VLAN A3 and the network B. In the policy tree, we can see that, in addition to the global address of A3, several host addresses (/32) appear. These addresses probably exist because some rules are related to them. Since we would like to know the global policies between the two networks, only the global addresses will be selected.

2522 different flows exist for this couple of addresses. In these flows are detected no upstream, 3270 downstream, 76 no-reply and 2401 useless-established anomalies. Such number of anomalies can naturally not be checked one by one by a human. However, to examine the policy applied for certain important flows can be interesting. For example, we immediately see that input ICMP packets in A are only accepted if there are sent in response to packets sent in the other direction, or if there are of the type «Fragmentation Needed» or «Time Exceeded». Let us look now at the TCP connections. If we want to see which connection is authorized from A3 to B, first we will select any global port (without a specific rule relating to it) as an A port. Any global port is suitable because the source port of a TCP connection is randomly selected. We see that the policy of A is to allow outgoing connections and to only accept replies of established flows. In the

[1]The match *ctorigdst* is not widely used. It matches based on the original destination IP specification of the *conntrack* entry that the packet is related to.[Andr]

firewall B, two behaviours can be distinguished: First, for some B destination ports, requests are dropped because only established flows are accepted. However, for another set of ports, requests are accepted. Users can therefore make connections from A3 to B servers with ports 22 (ssh), 53(dns), 515 (print spooler), 9100 (used by HP printers). It means that by default a host that opens an SSH port in the network B can receive connections from A.

The high number of *downstream blocked* anomalies is absolutely normal in this kind of network structure. Indeed, the general policy is *allow everything* for most of outgoing requests and *accept only some server ports* for the incoming ones. So, it is generally the downstream firewall that blocks the stream. The other anomaly that regularly occurs is the *useless established* one. This was also predictable since the policy that has just been explained involves that the rules that block the incoming requests reject only the incoming new connection but obviously accept the outgoing ones. The consequence is that a lot of established rules are encountered despite that most of them only reject packets. Note that nearly all the networks in our sample use the same policy.

We can assume that a category of networks does not apply the same policy: the ones that host servers. The networks C1 and VLAN A1 have been built up with this objective. Let us see what the policy is between A3, the same client network, and C1. More precisely, we will look at the policy between any host in A3 and a known server in C1. Two different policies are noticed: The first policy is the same as the one of the previous example, only incoming established connections are authorized. It is equivalent to drop the requests. The second one is about the authorized ports and it accepts the incoming requests as expected. Using DFA, the network administrator can have a quick view of the ports opened: 21 (ftp), 80 (www), 88 (kerberos) and 389 (ldap). In fact, the policy in C1 is not really different from the previous ones.

### 5.3.2 More relevant anomalies

As explained in the section 4.2.3, some anomalies have been classified as more relevant than the other ones. It is the anomalies that are related to rules containing an explicit reference to the other network. Unfortunately, in our sample, each network is independent from the other and does not have any rule concerning another network. The only exception is between the networks C1 and C2 where a few rules concern the other one.

Between these networks, there is actually one anomaly considered as relevant. This anomaly is about the firewall host in C2. The policy of C2 is to refuse all the connections to the bridge. In the other hand, C1 explicitly accepts all the packets from and to the network C2, including the firewall. Unfortunately, this anomaly is not a really good example of a interesting one since only one host is concerned by dropping and the accepting rule includes the entire network. The other rules between C1 and C2 accept packets in both directions with the other network. Obviously, these rules do not generate any anomaly.

### 5.3.3 Request limited between networks

Since the more relevant anomalies cannot be applied in our sample network, we will try to see if it is possible to find other interesting anomalies in the network. The policy applied in the subnets involves that only a few ports are accepted for incoming requests and only a few ports are refused for outgoing requests.

The reason why some incoming connections are authorized for some hosts and ports is that the requests to these servers are accepted from any foreign host. Therefore, it is curious that a

firewall of another network dropped the requests to this server. For instance, if a public FTP server is installed in one network, each other network should authorize requests to this server. This policy does not apply to the entire Internet but it does in a network as the sample where each network partly trusts the other ones.

How is it possible to detect easily this kind of anomaly? With the policy tree, the only thing to do is to explore the addresses for which we would like to check the anomaly. For simpler handling, the best choice is to select the client part as *A* and the server part as *B*. Then, for a given protocol (UDP or TCP), a general port has to be chosen for A. At this point, we will check whether some B port nodes contain *upstream blocked* anomalies. This anomaly means that packets are dropped by the source firewall while they are accepted by the destination one.

Tests have been performed between several networks of our sample. The table 5.2 summarises the results. In this table, when only the network code is specified, it means that the address checked is the general address of this network. The result is so the default behaviour of the network.

| Client Network | Server Network | Dst ports accepted for servers but refused for clients |
|---|---|---|
| B | C1 | - |
| B | C1, a server | UDP ports 379, 389, 636 (LDAP) |
| B | C2 | - |
| B | A3 | - |
| B | A1 | TCP ports 1214 (Kazaa), 1285, 1299, 1331, ... and UDP ports 0-52, 54-909, ... |
| A3 | B | - |
| A3 | C1, a server | - |

**Table 5.2 –** Summary of the test results

These results do not mean that there are problems in these networks. The second test as well as the fifth one show such results because the destination networks are very permissive networks and B has some restrictions for its output requests. Actually, this result should be quite comforting for the administrators of the different networks.

### 5.3.4 Other verifications

Using the policy tree, it is possible to invent a large number of verifications depending on the networks checked. If a policy in a network is to permit the access from the outside to a set of computers, it could be verified with this application that the policy is well observed. It is now up to the imagination of the administrators to find the right tests to apply to their networks. All the possible tests cannot be imagined but however the policy tree seems to be a serviceable tool to perform a lot of experiments in a short time.

# Conclusion

The main objective of this thesis was to develop an application capable of analysing distributed firewall configurations. At the end of this work, we think that the objective is fulfilled. The tool can indeed parse and interpret distributed firewall configuration files. In our case, *Netfilter/iptables* is the only firewall configuration language supported but the software has been designed to be easily extended to support other languages.

At the beginning of the thesis, it was not clear how far we would be able to go as the way of achieving each step was obscur. After further reflection, we decided to build a intermediate language that allows to split the implementation in two parts. The first part would parse the firewall configuration files to this intermediate language and the second one would analyse the configuration. This permits to reuse each part in another framework. The language is quite simple but can represent pretty well most of firewall configurations in most of languages. It can also be easily extended to support new firewall features.

The large number of features supported by the language involves that the parser has to take into account all the input firewall configuration language details. The parser is not only a widget that supports the ten basic matches. It is rather a tool that can be taken apart from the rest of the implementation and that offers enough flexibility to be used in other contexts. This parser has been implemented for the *Netfilter/iptables* configuration language but can be easily extended. For all these reasons, the time spent on this part has been more important than expected, it exceeded half of the implementation time.

Once it has been done, a new functionality has been added to our software. This is an analyser of distributed firewalls. This implementation has two main parts, the policy tree and the anomaly finder. The policy tree permits to explore the policies of each flow between two firewalls and so to find anomalies. A few ones of theses anomalies has been thoroughly defined and are automatically detected in the policy tree.

The graphical user interface has not been explained at all in this report. The parsing and the analyser can be controlled (it is even recommended) through this GUI developed in Java Swing/AWT. No special care has been given to the design of this interface but it permits to quickly obtain the different results. It is also very useful to browse the policy tree. A short manual has been written about this interface and is available in Appendix B.

## Future work

All along the conception of this application, special attention has be given to make the software as extensible as possible. Each part that has been described above can be used independently from each other. One of its objectives is to permit reusing part of this application for new researches on the same subject. At this state of the implementation further works can be performed in four main directions.

The first one is to extend the parser to support new firewall configuration languages. As explained in the first chapter, several languages for firewalls exist and new ones will appear. The software can be used with Cisco routers running Cisco IOS or PIX OS for example. The only thing to do is to extend the parser classes and to redefine the parts of the algorithm proper to the language.

Another possible direction for researches is to use the parser for other purposes. For instance, it would be possible to implement a tool that performs the parsing in the opposite direction to our parser. It would convert our language configuration files into real firewall configurations. If it is combined with our parser, it would permit to convert any firewall configuration from one language to another using our intermediate language.

A third idea would be to use the parser and the policy tree to perform other analyses. One example could be to implement an analyser for simple firewalls. The parser would not have to be altered and only the construction algorithm of the policy tree would have to be changed. It would permit to have a quick overview of what is accepted and dropped in a single firewall. An anomaly detection system could also be added as in our application.

A last idea is to simply continue the work performed in this thesis. As you would have noticed, only an introduction to anomaly detection has been made in this work. So it should be possible to find more efficient anomaly detection mechanisms that return more reliable results.

The objective of this work was not to make a commercial application but to build good foundations of a distributed firewall analysis system. The objective is fulfilled and so this contribution could be a good starting point for further work on this subject.

# Bibliography


[Al S 03] E. Al-Shaer and H. Hamed. "Firewall Policy Advisor for Anomaly Detection and Rule Editing". *IEEE/IFIP Integrated Management (IM'2003)*, March 2003.

[Al S 04a] E. Al-Shaer and H. Hamed. "Discovery of Policy Anomalies in Distributed Firewalls". *IEEE INFOCOM'2004*, March 2004.

[Al S 04b] E. Al-Shaer and H. Hamed. "Modeling and Management of Firewall Policies". *IEEE Transactions on Network and System Management*, Vol. 1-1, April 2004.

[Andr] O. Andreasson. "Iptables-tutorial". http://iptables-tutorial.frozentux.net/.

[Bart 99] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. "Firmato: A Novel Firewall Management Toolkit". *Proceedings of 1999 IEEE Symposium on Security and Privacy*, May 1999.

[Bell 99] S. M. Bellovin. "Distributed firewalls". *;login:*, pp. 39–47, November 1999.

[Bone 02] J. Boney. *Cisco IOS in a Nutshell - A desktop quick reference for IOS on IP networks*. O'Reilly, 2002.

[Ches 03] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet Security - Second edition*. Addison-Wesley, 2003.

[Cisco] "Cisco Documentation". http://www.cisco.com/univercd/.

[Grid] F. Gridelet. *Commentaires coupe-feu*. UCL/SGSI/SISY, http://www.sisy.ucl.ac.be/index.php?blk=doc01&tag=more1.

[Hame 05] H. Hamed, E. Al-Shaer, and W. Marrero. "Modeling and Verification of IPSec and VPN Security Policies". *IEEE International Conference on Network Protocols (ICNP '05)*, pp. 259–278, 2005.

[Haro 02a] E. R. Harold and W. S. Means. *XML in a nutshell, 2nd Edition*. O'Reilly, 2002.

[Haro 02b] E. R. Harold. *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP and TrAX*. Addison-Wesley, 2002.

[Lewi 04] J. Lewis and W. Loftus. *Java Software Solutions 4th Edition: Foundations of Program Design*. Addison-Wesley, 2004.

[Maye 00] A. Mayer, A. Wool, and E. Ziskind. "Fang: A firewall analysis engine". *Proceedings of IEEE Computer Society Symposium on Security and Privacy*, pp. 177–187, May 2000.

[McLa 01] B. McLaughlin. *Java & XML, 2nd Edition: Solutions to Real-World Problems*. Addison-Wesley, 2001.

[Netfilt] "The Netfilter/iptables Project". http://www.netfilter.org/.

[Palm 04] B. Palmer and J. Nazario. *Secure Architectures with OpenBSD*, Chap. 22. Addison-Wesley, 2004.

[Post 81] J. Postel. "Transmission Control Protocol (TCP)". September 1981. RFC 793.

[Scha 04] F. Schanda. *Visualisation of iptables*. PhD thesis, Department of Computer Sciences - University of Bath, July 2004.

[Seda 01] J. Sedayao. *Cisco IOS access lists*. O'Reilly, 2001.

[W3C] "W3C - World Wide Web Consortium". http://www.w3.org/.

[Zwic 00] E. D. Zwicky, S. Cooper, and D. B. Chapman. *Building Internet Firewalls, Second Edition*. O'Reilly, 2000.

Appendix A

# Examples and test sets

## A.1 A simple example of `iptables-save` outputs

```
# Generated by iptables-save v1.2.9 on Thu May 25 00:00:34 2006
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT DROP [0:0]
:LOG_DROP - [0:0]
-A INPUT -i lo -j ACCEPT
-A INPUT -i eth1 -p tcp --dport 22 -j ACCEPT
-A INPUT -j LOG_DROP
-A FORWARD -s 192.168.0.0/255.255.255.0 -i eth1 -o eth0 -j ACCEPT
-A FORWARD -d 192.168.0.0/255.255.255.0 -i eth0 -o eth1 -j ACCEPT
-A FORWARD -j LOG_DROP
-A OUTPUT -o lo -j ACCEPT
-A OUTPUT -o eth1 -p tcp --sport 22 -j ACCEPT
-A OUTPUT -j LOG_DROP
-A LOG_DROP -j LOG --log-prefix '[IPTABLES DROP] : '
-A LOG_DROP -j DROP
COMMIT
# Completed on Thu May 25 00:00:34 2006
# Generated by iptables-save v1.2.9 on Thu May 25 00:00:34 2006
*mangle
:PREROUTING ACCEPT [9106515:6406097707]
:INPUT ACCEPT [168822:34496923]
:FORWARD ACCEPT [8846009:6341422559]
:OUTPUT ACCEPT [300064:39869273]
:POSTROUTING ACCEPT [8848118:6341656508]
:outtos - [0:0]
:pretos - [0:0]
-A PREROUTING -j pretos
-A OUTPUT -j outtos
-A outtos -p tcp -m tcp --dport 22 -j TOS --set-tos 0x10
-A outtos -p tcp -m tcp --sport 22 -j TOS --set-tos 0x10
COMMIT
# Completed on Thu May 25 00:00:34 2006
# Generated by iptables-save v1.2.9 on Thu May 25 00:00:34 2006
*nat
:PREROUTING ACCEPT [281192:40984704]
:POSTROUTING ACCEPT [181278:27591165]
:OUTPUT ACCEPT [245645:30059474]
-A PREROUTING -i eth0 -j DNAT --to-destination 192.168.0.5
-A POSTROUTING -s 192.168.0.0/255.255.255.0 -o eth0 -j MASQUERADE
COMMIT
# Completed on Thu May 25 00:00:34 2006
```

## A.2 Test sets for the parser

Here are a few test sets for the parser. A lot of tests have also been done with the real firewall configuration files.

### A.2.1 Rule merging

#### Merging of IP addresses

The input file:

```
# Generated by iptables-save v1.3.3 on Mon Mar 20 18:07:18 2006
*nat
:PREROUTING ACCEPT [3506:371499]
:POSTROUTING ACCEPT [523:35418]
:OUTPUT ACCEPT [523:35418]
COMMIT
# Completed on Mon Mar 20 18:07:18 2006
# Generated by iptables-save v1.3.3 on Mon Mar 20 18:07:18 2006
*filter
:INPUT DROP [3505:370891]
:FORWARD DROP [0:0]
:OUTPUT DROP [10:520]
:ch1 - [0:0]
:ch2 - [0:0]
-A FORWARD -m iprange --src-range 192.168.1.13-192.168.1.15 -j ch1
-A ch1 -s ! 192.168.1.16 -j ch2
-A ch1 -j ACCEPT
-A ch2 -s 192.168.1.14 -j DROP
COMMIT
# Completed on Mon Mar 20 18:07:18 2006
```

The result:

```
<?xml version="1.0" encoding="UTF-8"?>
<firewall type="Iptables (iptables-save)">
  <filtertable>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
      <ipv4>
        <src inverted="false">
          <ip>192.168.1.14</ip>
        </src>
      </ipv4>
    </frule>
    <frule target="ACCEPT">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
      <ipv4>
        <src inverted="false">
          <ip>192.168.1.14</ip>
          <mask>255.255.255.254</mask>
        </src>
      </ipv4>
    </frule>
    <frule target="ACCEPT">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
      <ipv4>
        <src inverted="false">
          <ip>192.168.1.13</ip>
          <mask>255.255.255.255</mask>
```

```
        </src>
      </ipv4>
    </frule>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfaceout inverted="false">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfacein inverted="false">firewall_host</interfacein>
    </frule>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>x.x.x.x</ip>
      <mask>x.x.x.x</mask>
    </network>
  </interface>
</firewall>
```

### Merging of port ranges

The input file:

```
# Generated by iptables-save v1.3.3 on Mon Mar 20 18:07:18 2006
*nat
:PREROUTING ACCEPT [3506:371499]
:POSTROUTING ACCEPT [523:35418]
:OUTPUT ACCEPT [523:35418]
COMMIT
# Completed on Mon Mar 20 18:07:18 2006
# Generated by iptables-save v1.3.3 on Mon Mar 20 18:07:18 2006
*filter
:INPUT DROP [3505:370891]
:FORWARD DROP [0:0]
:OUTPUT DROP [10:520]
:ch1 - [0:0]
:ch2 - [0:0]
-A FORWARD -p tcp -m tcp --sport 20:40 -j ch1
-A ch1 -p tcp -m tcp ! --sport 25:35 -j ch2
-A ch1 -j ACCEPT
-A ch2 -p tcp -m tcp --sport 0:40 -j DROP
COMMIT
# Completed on Mon Mar 20 18:07:18 2006
```

The result:

```
<?xml version="1.0" encoding="UTF-8"?>
<firewall type="Iptables (iptables-save)">
  <filtertable>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
      <protocol inverted="false">tcp</protocol>
      <tcp>
        <srcport inverted="false">
          <range from="20" to="40" />
        </srcport>
        <srcport inverted="true">
          <range from="25" to="35" />
        </srcport>
      </tcp>
    </frule>
```

```
    <frule target="ACCEPT">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
      <protocol inverted="false">tcp</protocol>
      <tcp>
        <srcport inverted="false">
          <range from="20" to="40" />
        </srcport>
      </tcp>
    </frule>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfaceout inverted="false">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfacein inverted="false">firewall_host</interfacein>
    </frule>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>x.x.x.x</ip>
      <mask>x.x.x.x</mask>
    </network>
  </interface>
</firewall>
```

### Merging of ICMP properties

The input file:

```
# Generated by iptables-save v1.3.3 on Mon Mar 20 18:07:18 2006
*nat
:PREROUTING ACCEPT [3506:371499]
:POSTROUTING ACCEPT [523:35418]
:OUTPUT ACCEPT [523:35418]
COMMIT
# Completed on Mon Mar 20 18:07:18 2006
# Generated by iptables-save v1.3.3 on Mon Mar 20 18:07:18 2006
*filter
:INPUT DROP [3505:370891]
:FORWARD DROP [0:0]
:OUTPUT DROP [10:520]
:ch1 - [0:0]
:ch2 - [0:0]
-A FORWARD -p icmp -m icmp --icmp-type 8 -j ch1
-A ch1 -p icmp -m icmp ! --icmp-type 8/2 -j ch2
-A ch1 -j ACCEPT
-A ch2 -p icmp -m icmp --icmp-type 4 -j DROP
-A ch2 -p icmp -m icmp --icmp-type 8/3 -j DROP
-A ch2 -p icmp -m icmp --icmp-type 8 -j DROP
COMMIT
# Completed on Mon Mar 20 18:07:18 2006
```

The result:

```
<?xml version="1.0" encoding="UTF-8"?>
<firewall srclang="Iptables (iptables-save)">
  <filtertable>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
      <protocol inverted="false">icmp</protocol>
```

```
      <icmp>
        <type inverted="false" value="8" code="3" />
      </icmp>
    </frule>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
      <protocol inverted="false">icmp</protocol>
      <icmp>
        <type inverted="false" value="8" />
        <type inverted="true" value="8" code="2" />
      </icmp>
    </frule>
    <frule target="ACCEPT">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
      <protocol inverted="false">icmp</protocol>
      <icmp>
        <type inverted="false" value="8" />
      </icmp>
    </frule>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfaceout inverted="false">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfacein inverted="false">firewall_host</interfacein>
    </frule>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>x.x.x.x</ip>
      <mask>x.x.x.x</mask>
    </network>
  </interface>
</firewall>
```

### Merging of SPI ranges

The input file:

```
# Generated by iptables-save v1.3.3 on Mon Mar 20 18:07:18 2006
*nat
:PREROUTING ACCEPT [3506:371499]
:POSTROUTING ACCEPT [523:35418]
:OUTPUT ACCEPT [523:35418]
COMMIT
# Completed on Mon Mar 20 18:07:18 2006
# Generated by iptables-save v1.3.3 on Mon Mar 20 18:07:18 2006
*filter
:INPUT DROP [3505:370891]
:FORWARD DROP [0:0]
:OUTPUT DROP [10:520]
:ch1 - [0:0]
:ch2 - [0:0]
-A FORWARD -p ah -m ah --ahspi 500:1000000000 -j ch1
-A ch1 -p ah -m ah --ahspi ! 1000:1500 -j ch2
-A ch1 -j ACCEPT
-A ch2 -p ah -m ah --ahspi 300:1750 -j DROP
COMMIT
# Completed on Mon Mar 20 18:07:18 2006
```

The result:

```
<?xml version="1.0" encoding="UTF-8"?>
<firewall type="Iptables (iptables-save)">
  <filtertable>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
      <protocol inverted="false">ah</protocol>
      <ah>
        <spi inverted="false">
          <range from="500" to="1750" />
        </spi>
        <spi inverted="true">
          <range from="1000" to="1500" />
        </spi>
      </ah>
    </frule>
    <frule target="ACCEPT">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
      <protocol inverted="false">ah</protocol>
      <ah>
        <spi inverted="false">
          <range from="500" to="1000000000" />
        </spi>
      </ah>
    </frule>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfaceout inverted="false">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfacein inverted="false">firewall_host</interfacein>
    </frule>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>x.x.x.x</ip>
      <mask>x.x.x.x</mask>
    </network>
  </interface>
</firewall>
```

## A.3 Test sets for the analyser

Here are a few test sets for the analyser. A lot of tests have also been done with the real firewall configuration files.

### A.3.1 Recognition of the firewall positioning (see Figure 1.3)

#### Configuration of the firewall A in *face-to-face* positioning

```
<?xml version="1.0" encoding="UTF-8"?>
<firewall type="Iptables (iptables-save)">
  <filtertable>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>2.0.0.1</ip>
    </network>
```

```
    <network>
      <ip>2.0.0.2</ip>
    </network>
  </interface>
  <interface name="intA">
    <network>
      <ip>2.0.0.0</ip>
      <mask>255.0.0.0</mask>
    </network>
  </interface>
  <interface name="exterA">
    <network>
      <ip>0.0.0.0</ip>
      <mask>0.0.0.0</mask>
    </network>
  </interface>
</firewall>
```

### Configuration of the firewall B in *face-to-face* positioning

```
<?xml version="1.0" encoding="UTF-8"?>
<firewall type="Iptables (iptables-save)">
  <filtertable>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>1.0.0.1</ip>
    </network>
    <network>
      <ip>1.0.0.2</ip>
    </network>
  </interface>
  <interface name="exterB">
    <network>
      <ip>0.0.0.0</ip>
      <mask>0.0.0.0</mask>
    </network>
  </interface>
  <interface name="intB">
    <network>
      <ip>1.0.0.0</ip>
      <mask>255.0.0.0</mask>
    </network>
  </interface>
</firewall>
```

### Configuration of the firewall A in *one firewall behind the second one* positioning

```
<?xml version="1.0" encoding="UTF-8"?>
<firewall type="Iptables (iptables-save)">
  <filtertable>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>1.0.0.1</ip>
    </network>
    <network>
      <ip>1.0.0.2</ip>
    </network>
  </interface>
  <interface name="intA">
    <network>
      <ip>1.0.0.0</ip>
      <mask>255.0.0.0</mask>
```

```
    </network>
  </interface>
  <interface name="exterA">
    <network>
      <ip>0.0.0.0</ip>
      <mask>0.0.0.0</mask>
    </network>
  </interface>
</firewall>
```

### Configuration of the firewall B in *one firewall behind the second one* positioning

```
<?xml version="1.0" encoding="UTF-8"?>
<firewall type="Iptables (iptables-save)">
  <filtertable>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>1.1.0.1</ip>
    </network>
    <network>
      <ip>1.1.0.2</ip>
    </network>
  </interface>
  <interface name="exterB">
    <network>
      <ip>0.0.0.0</ip>
      <mask>0.0.0.0</mask>
    </network>
  </interface>
  <interface name="intB">
    <network>
      <ip>1.1.0.0</ip>
      <mask>255.255.0.0</mask>
    </network>
  </interface>
</firewall>
```

## A.3.2 Finding more relevant anomaly

### Example 1 containing a more relevant anomaly

```
<?xml version="1.0" encoding="UTF-8"?>
<firewall type="Iptables (iptables-save)">
  <filtertable>
     <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfaceout inverted="false">firewall_host</interfaceout>
    </frule>
    <frule target="ACCEPT">
      <interfacein inverted="false">firewall_host</interfacein>
    </frule>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>1.1.36.119</ip>
    </network>
  </interface>
  <interface name="eth2">
    <network>
```

```
      <ip>0.0.0.0</ip>
      <mask>0.0.0.0</mask>
    </network>
  </interface>
  <interface name="eth1">
    <network>
      <ip>1.1.36.0</ip>
      <mask>255.255.252.0</mask>
    </network>
  </interface>
</firewall>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<firewall type="Iptables (iptables-save)">
  <filtertable>
    <frule target="ACCEPT">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
      <ipv4>
        <dst inverted="false">
          <ip>1.1.36.0</ip>
          <mask>255.255.252.0</mask>
        </dst>
        <src inverted="false">
          <ip>1.1.228.1</ip>
          <mask>255.255.255.255</mask>
        </src>
      </ipv4>
    </frule>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfaceout inverted="false">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfacein inverted="false">firewall_host</interfacein>
    </frule>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>1.1.233.234</ip>
    </network>
    <network>
      <ip>1.1.228.69</ip>
    </network>
  </interface>
  <interface name="eth0">
    <network>
      <ip>0.0.0.0</ip>
      <mask>0.0.0.0</mask>
    </network>
  </interface>
  <interface name="eth1.161">
    <network>
      <ip>1.1.228.0</ip>
      <mask>255.255.255.128</mask>
    </network>
  </interface>
</firewall>
```

### Example 2 containing a more relevant anomaly

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<firewall type="Iptables (iptables-save)">
  <filtertable>
    <frule target="ACCEPT">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
      <ipv4>
        <src inverted="false">
          <ip>1.1.36.0</ip>
          <mask>255.255.255.0</mask>
        </src>
        <dst inverted="false">
          <ip>1.1.228.1</ip>
          <mask>255.255.255.255</mask>
        </dst>
      </ipv4>
    </frule>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfaceout inverted="false">firewall_host</interfaceout>
    </frule>
    <frule target="ACCEPT">
      <interfacein inverted="false">firewall_host</interfacein>
    </frule>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>1.1.36.119</ip>
    </network>
  </interface>
  <interface name="eth2">
    <network>
      <ip>0.0.0.0</ip>
      <mask>0.0.0.0</mask>
    </network>
  </interface>
  <interface name="eth1">
    <network>
      <ip>1.1.36.0</ip>
      <mask>255.255.252.0</mask>
    </network>
  </interface>
</firewall>


<?xml version="1.0" encoding="UTF-8"?>
<firewall type="Iptables (iptables-save)">
  <filtertable>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfaceout inverted="false">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfacein inverted="false">firewall_host</interfacein>
    </frule>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>1.1.233.234</ip>
    </network>
    <network>
      <ip>1.1.228.69</ip>
    </network>
  </interface>
  <interface name="eth0">
```

```
    <network>
      <ip>0.0.0.0</ip>
      <mask>0.0.0.0</mask>
    </network>
  </interface>
  <interface name="eth1.161">
    <network>
      <ip>1.1.228.0</ip>
      <mask>255.255.255.128</mask>
    </network>
  </interface>
</firewall>
```

### Example 3 containing a more relevant anomaly

```
<?xml version="1.0" encoding="UTF-8"?>
<firewall type="Iptables (iptables-save)">
  <filtertable>
    <frule target="ACCEPT">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfaceout inverted="false">firewall_host</interfaceout>
    </frule>
    <frule target="ACCEPT">
      <interfacein inverted="false">firewall_host</interfacein>
    </frule>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>1.1.36.119</ip>
    </network>
  </interface>
  <interface name="eth2">
    <network>
      <ip>0.0.0.0</ip>
      <mask>0.0.0.0</mask>
    </network>
  </interface>
  <interface name="eth1">
    <network>
      <ip>1.1.36.0</ip>
      <mask>255.255.252.0</mask>
    </network>
  </interface>
</firewall>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<firewall type="Iptables (iptables-save)">
  <filtertable>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
      <ipv4>
        <dst inverted="false">
          <ip>1.1.36.0</ip>
          <mask>255.255.252.0</mask>
        </dst>
        <src inverted="false">
          <ip>1.1.228.1</ip>
          <mask>255.255.255.255</mask>
        </src>
      </ipv4>
    </frule>
```

```
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfaceout inverted="false">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfacein inverted="false">firewall_host</interfacein>
    </frule>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>1.1.233.234</ip>
    </network>
    <network>
      <ip>1.1.228.69</ip>
    </network>
   </interface>
  <interface name="eth0">
    <network>
      <ip>0.0.0.0</ip>
      <mask>0.0.0.0</mask>
    </network>
  </interface>
  <interface name="eth1.161">
    <network>
      <ip>1.1.228.0</ip>
      <mask>255.255.255.128</mask>
    </network>
  </interface>
</firewall>
```

### Example that does not contain a more relevant anomaly

```
<?xml version="1.0" encoding="UTF-8"?>
<firewall type="Iptables (iptables-save)">
  <filtertable>
    <frule target="ACCEPT">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfaceout inverted="false">firewall_host</interfaceout>
    </frule>
    <frule target="ACCEPT">
      <interfacein inverted="false">firewall_host</interfacein>
    </frule>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>1.1.36.119</ip>
    </network>
  </interface>
  <interface name="eth2">
    <network>
      <ip>0.0.0.0</ip>
      <mask>0.0.0.0</mask>
    </network>
  </interface>
  <interface name="eth1">
    <network>
      <ip>1.1.36.0</ip>
      <mask>255.255.252.0</mask>
    </network>
  </interface>
</firewall>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<firewall type="Iptables (iptables-save)">
  <filtertable>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
      <ipv4>
        <dst inverted="false">
          <ip>1.1.0.0</ip>
          <mask>255.255.255.0</mask>
        </dst>
        <src inverted="false">
          <ip>1.1.228.1</ip>
          <mask>255.255.255.255</mask>
        </src>
      </ipv4>
    </frule>
    <frule target="DROP">
      <interfacein inverted="true">firewall_host</interfacein>
      <interfaceout inverted="true">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfaceout inverted="false">firewall_host</interfaceout>
    </frule>
    <frule target="DROP">
      <interfacein inverted="false">firewall_host</interfacein>
    </frule>
  </filtertable>
  <interface name="firewall_host">
    <network>
      <ip>1.1.233.234</ip>
    </network>
    <network>
      <ip>1.1.228.69</ip>
    </network>
  </interface>
  <interface name="eth0">
    <network>
      <ip>0.0.0.0</ip>
      <mask>0.0.0.0</mask>
    </network>
  </interface>
  <interface name="eth1.161">
    <network>
      <ip>1.1.228.0</ip>
      <mask>255.255.255.128</mask>
    </network>
  </interface>
</firewall>
```

Appendix B

# Short manual for DFA

The GUI of DFA is quite intuitive to use. The following sections provide some brief explanations of how to use it.

## B.1 To launch the application

Read the *README* file in the directory of the application.

## B.2 Overview of the interface

The main window of DFA is represented at Figure B.1. It contains a menu bar and a desktop. The menu allows to close the application or to launch a new parsing or analysis window. When the application is launched, only a parsing window is displayed.

## B.3 The parser

The full parsing frame is shown at Figure B.2. The steps of parsing are the following:

- Choose the firewall configuration language. (For the moment, only *iptables* is supported)
- Select a «Firewall input file». This is the firewall configuration file in the language defined.
- If you want a empty interface list file to be generated, check the box and define a file for interface list in the area below.
- When the fields are filled, the «Analyze» button becomes available. The analysis reads and parses all the rules of the input file. It also generates the interface list file if asked. The result of this parsing is printed in the message box below. The log-rules dropped and the matches found that are not supported are also displayed.
- If the interface list has been generated, it has to be filled now. The way to fill it is explained in 3.2.3.

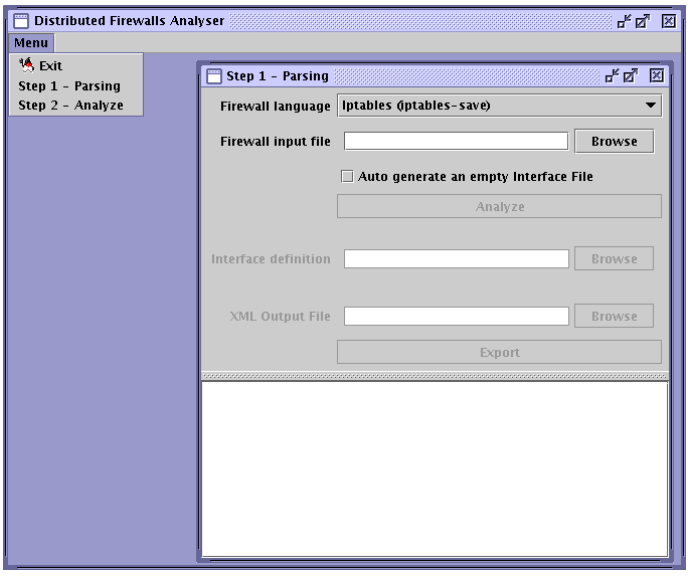



**Figure B.1 –** The entire DFA window

- When the output and the interface files are mentioned, the «Export» button becomes available. It outputs the entire XML document in our intermediate language. The result is printed in the message box.

## B.4 The analyser

Figure B.3 shows an analyser frame. The different analysis steps are the following:

- The two first fields have to contain the XML firewall configuration files. The first firewall is consider as the firewall A and the second as the firewall B.
- When the «Analyze» button has been pressed, the first level of the tree is built.
- Then you have to select either a network of A to explore or «All». If you choose «All», all the nodes of the tree are explored. In medium firewall, it may need more than 1GByte of memory and some minutes to process. So, this is not advised except for small test firewalls with a few rules. If you have not chosen «All», you have to select a B network.
- The policy tree is displayed. You can explore it and select leaf nodes to see the policy. The numbers appended to the node names are: *(the number of upstream-blocked anomalies, the number of downstream-blocked anomalies, the number of no-reply anomalies, the number of useless-established anomalies) [The number of more relevant anomalies].* If the node is not a leaf node, these numbers are the sum of those values for their children.
- The two policy tables displayed below correspond to the policy table of A (left) and to the policy table of B (right). The general policy found is summarised as well as the anomalies. Then, the rules concerning this flow in each direction are displayed. The last rule is always a rule that matches all the packet of this flow.

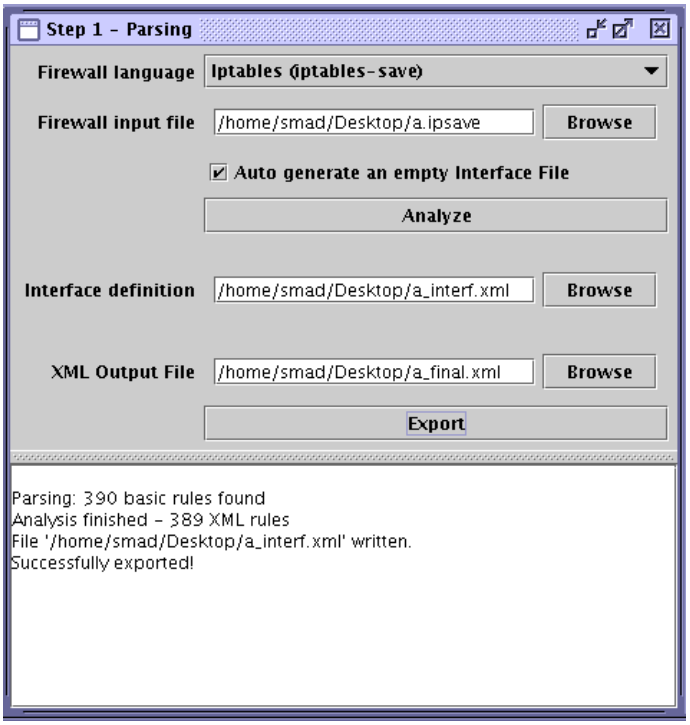



**Figure B.2 –** A successful parsing

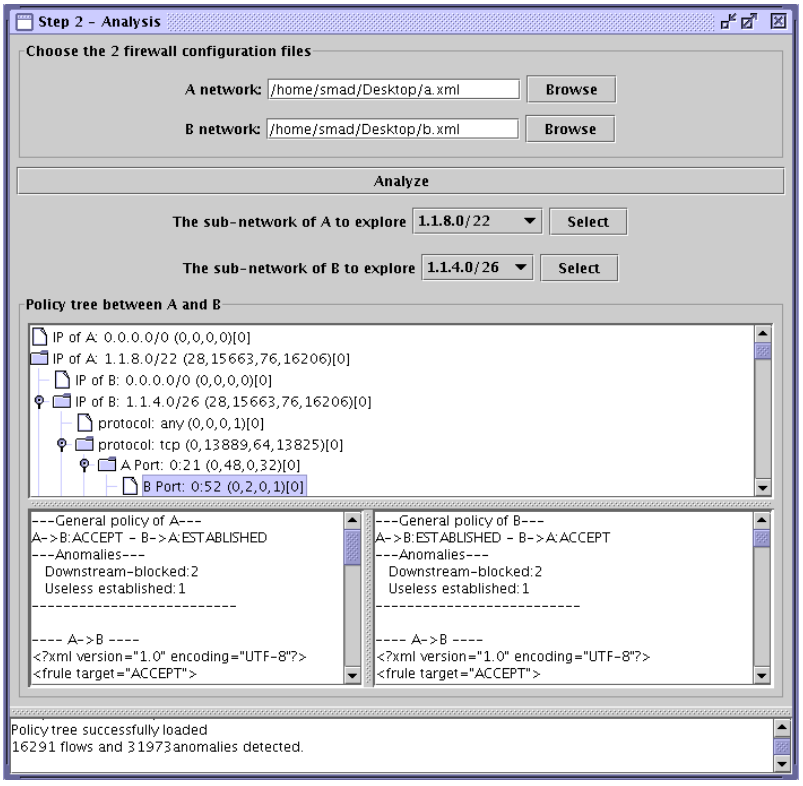



**Figure B.3 –** An analyser frame