Management of Exceptions on Access Control Policies*

J. Garcia-Alfaro[1,2], F. Cuppens[1], and N. Cuppens-Boulahia[1]

[1] GET/ENST-Bretagne, 35576 Cesson Sévigné - France,
{frederic.cuppens,nora.cuppens}@enst-bretagne.fr

[2] Universitat Oberta de Catalunya, 08018 Barcelona - Spain,
joaquin.garcia-alfaro@uoc.edu

Abstract. The use of languages based on positive or negative expressiveness is very common for the deployment of security policies (i.e., deployment of permissions and prohibitions on firewalls through single-handed positive or negative condition attributes). Although these languages may allow us to specify any policy, the single use of positive or negative statements alone leads to complex configurations when excluding some specific cases of general rules that should always apply. In this paper we survey such a management and study existing solutions, such as ordering of rules and segmentation of condition attributes, in order to settle this lack of expressiveness. We then point out to the necessity of full expressiveness for combining both negative and positive conditions on firewall languages in order to improve this management of exceptions on access control policies. This strategy offers us a more efficient deployment of policies, even using fewer rules.

1 Introduction

Current firewalls are still being configured by security officers in a manual fashion. Each firewall usually provides, moreover, its own configuration language that, most of the times, present a lack of expressiveness and semantics. For instance, most firewall languages are based on rules in the form $R_i : \{condition_i\} \rightarrow decision_i$, where i is the relative position of the rule within the set of rules, $decision_i$ is a boolean expression in $\{accept, deny\}$, and $\{condition_i\}$ is a conjunctive set of condition attributes, such as *protocol* (p), *source* (s), *destination* (d), *source port* (sport), *destination port* (dport), and so on. This conjunctive set of conditions attributes, i.e., $\{condition_i\}$, is mainly composed of either positive (e.g., A) or negative (e.g., $\neg A$) statements for each attribute, but does not allow us to combine both positive and negative statements (e.g., $A \wedge \neg B$) for a single attribute, as many other languages with

* This work was supported by funding from the French ministry of research, under the *ACI DESIRS* project; and the Spanish Government (CICYT) project *SEG2004-04352-C04-04*.

full expressive power, such as SQL-like languages [9], do. The use of more general access control languages, such as the eXtensible Access Control Markup Language (XACML) [11], also present such a lack of expressiveness. This fact leads to complex administration tasks when dealing with exclusion issues on access control scenarios, i.e., when some cases must be excluded of general rules that should always apply.

Let us suppose, for instance, the policy of a hospital where, in general, all doctors are allowed to consult patient's medical records. Later, the policy changes and doctors going on strike are not allowed to consult medical records; but, as an exception to the previous one, and for emergencies purposes, doctors going on strike are still allowed to consult the records. Regarding the use of a language with expressiveness enough to combine both positive and negative statements, one may deploy the previous example as follows. We first assume the following definitions: *(A)* "*Doctors*"; *(B)* "*Doctors going on strike*"; *(C)* "*Doctors working on emergencies*". We then deploy the hospital's policy goals, i.e., *(1)* "*In Hospital, doctors can access patient's medical records.*"; *(2)* "*In Hospital, and only for emergency purposes, doctors going on strike can access patient's medical records.*"; through the following statement: "*In Hospital, $(A \wedge (\neg B \vee C))$ can access patient's medical records*".

The use of languages based on partial expressiveness may lead us to very complicated situations when managing this kind of configurations on firewalls and filtering routers. In this paper, we focus on this problem and survey current solutions, such as first and last matching strategies, segmentation of condition attributes, and partial ordering of rules. We then discuss how the combination of both negative and positive expressiveness on configuration languages may help us to improve those solutions. This strategy allows to perform a more efficient deployment of network access control policies, even using fewer rules, and properly manage exceptions and exclusion of attributes on firewall and filtering router configurations.

The rest of this paper is organized as follows. Section 2 recalls our motivation problem, by showing some representative examples, surveying related solutions, and overviewing their advantages and drawbacks. Section 3 then discusses our approach. Section 4 overviews some related work, and, finally, Section 5 closes the paper.

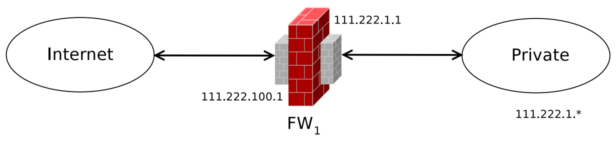
2 Management of Exceptions via Partial Expressiveness

Before going further in this section, let us start with an example to illustrate our motivation problem. We first consider the network setup shown in Figure 1(a), together with the following general premise: "*In Private, all hosts can access web resources on the Internet*". We assume, moreover, that firewall FW_1 implements a closed default policy, specified in its set of rules at the last entry, in the form $R_n : deny$. Then, we deploy the premise over firewall FW_1 with the following rule:

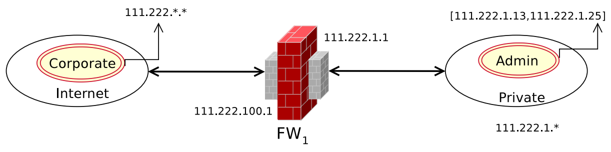
$$\boxed{R_1 : (s \in Private \wedge d \in any \wedge p = tcp \wedge dport = 80) \rightarrow accept}$$

Regarding the exclusion issues pointed out above, and according to the extended setup shown in Figure 1(b), let us assume that we must now apply the following three exceptions over the general security policy:

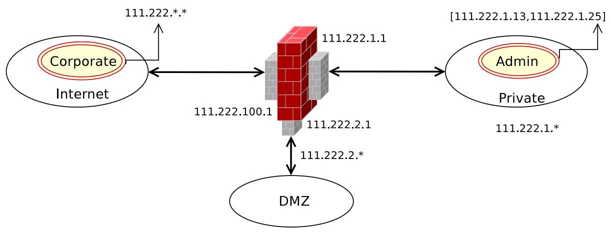
1. *The interfaces of firewall FW_1 (i.e., Interf-fw = {111.222.1.1, 111.222.100.1}) are not allowed to access web resources on the Internet.*
2. *The hosts in Admin are not allowed to access web resources.*
3. *The hosts in Corporate do not belong to the zone Internet.*



(a) Simple access control policy.



(b) Same policy with some excluded zones.



(c) Extended access control policy.

Fig. 1. Sample access control policy setups.

According to the first exception, we should exclude the IP address 111.222.1.1 from the hosts of *Private*. Similarly, we must exclude the whole set of hosts in zone *Admin* from the zone *Private*, and the whole set of hosts in zone *Corporate*, i.e., the range $111.222.*.*$, from *Internet*. The use of a language with expressiveness enough to combine both positive and negative statements may allow us to deploy the previous policy goal, i.e., "*All the hosts in (Private* $\wedge$ *¬Admin* $\wedge$ *¬Interf-fw) are allowed to access web resources on (Internet* $\wedge$ *¬Corporate)*", as the following single rule:

$$\boxed{R_1 : (s \in (\text{Private} \wedge \neg\text{Admin} \wedge \neg\text{Interf-fw}) \wedge d \in \ (\text{any} \wedge \neg\text{Corporate}) \wedge p = tcp \wedge dport = 80) \rightarrow accept}$$

However, the lack of semantics and expressiveness of current firewall configuration languages (specially the impossibility for combining both positive and negative statements on single condition attributes) forces us to use different strategies to make up for this lack of expressiveness. We overview in the following sections some possible solutions for applying the previous example by means of such languages.

2.1 First Matching Strategy

Most firewalls solve the managing of exceptions by an ordering of rules. For instance, the configuration language for IPTables, the administration software used to configure GNU/Linux-based firewalls through the Netfilter framework, is based on a *first matching* strategy, i.e., the firewall is parsing the rules until a rule applies. When no rule applies, the decision depends on the default policy: in the case of an open policy, the packet is accepted whereas if the policy is closed, the packet is rejected. Other languages, like the configuration language of IPFilter, the administration software for configuring FreeBSD-, NetBSD- and Solaris 10-based firewalls, apply the opposite strategy, called *last matching*. Similar approaches have also been proposed in other security domains, such as the formal access control proposed in [10] to specify protection policies on XML databases. Through a first matching strategy, one may specify the handling of exceptions in the form $R_1 : (s \in (A \wedge \neg\, B)) \rightarrow accept$ by means of the following ordering of rules:

$$\boxed{\begin{array}{l} R_1 : (s \in B) \rightarrow deny \\ R_2 : (s \in A) \rightarrow accept \end{array}}$$

Regarding the access control setup shown in Figure 1(b), together with the set of policy goals and exceptions defined above, i.e., "*All the hosts in (Private $\wedge$ ¬Admin $\wedge$ ¬Interf-fw) are allowed to access web resources on (Internet $\wedge$ ¬Corporate)*", a possible solution for such a motivation example through a first matching strategy shall be the following set of rules:

$$\boxed{\begin{array}{l} R_1 : (s \in 111.222.1.0/24 \wedge d \in 111.222.0.0/16 \wedge p = tcp \wedge dport = 80) \rightarrow deny \\ R_2 : (s \in [111.222.1.13, 111.222.1.25] \wedge d \in any \wedge p = tcp \wedge dport = 80) \rightarrow deny \\ R_3 : (s \in 111.222.1.1 \wedge d \in any \wedge p = tcp \wedge dport = 80) \rightarrow deny \\ R_4 : (s \in 111.222.1.0/24 \wedge d \in any \wedge p = tcp \wedge dport = 80) \rightarrow accept \\ R_5 : deny \end{array}}$$

Although this strategy offers a proper solution for the handling of exceptions, it is well known that it may introduce many other configuration errors, such as *shadowing* of rules and *redundancy* [4, 2], as well as important drawbacks when managing rule updates, specially when adding or removing new general rules and/or exceptions. For example, if we consider now the extended access control policy shown in Figure 1(c), together with the insertion of the following general rule: "*In Private, all hosts can access web resources on the zone DMZ*"; and the insertion of the following exception to the previous rule: "*The interfaces of firewall FW$_1$ (i.e., Interf-fw = {111.222.1.1, 111.222.2.1, 111.222.100.1}) are not allowed to access web resources on the zone DMZ*"; we shall agree that the resulting rules according with these two new premises are the following ones: $R_i : (s \in 111.222.1.1 \wedge d \in 111.222.2.0/24 \wedge p = tcp \wedge dport = 80) \rightarrow deny; R_j : (s \in 111.222.1.0/24 \wedge d \in 111.222.2.0/24 \wedge p = tcp \wedge dport = 80) \rightarrow accept$. Such new rules must be inserted in the previous set of rules as shown in Figure 2.

Notice that, in the previous example, the only possible ordering of rules that guarantees the defined assumptions forces us to place the new general rule in the second

$R_1 : (s \in 111.222.1.1 \land d \in \mathit{any} \land p = \mathit{tcp} \land \mathit{dport} = 80) \rightarrow \mathit{deny}$
$R_2 : (s \in 111.222.1.0/24 \land d \in 111.222.2.0/24 \land p = \mathit{tcp} \land \mathit{dport} = 80) \rightarrow \mathit{accept}$
$R_3 : (s \in [111.222.1.13, 111.222.1.25] \land d \in \mathit{any} \land p = \mathit{tcp} \land \mathit{dport} = 80) \rightarrow \mathit{deny}$
$R_4 : (s \in 111.222.1.0/24 \land d \in 111.222.0.0/16 \land p = \mathit{tcp} \land \mathit{dport} = 80) \rightarrow \mathit{deny}$
$R_5 : (s \in 111.222.1.0/24 \land d \in \mathit{any} \land p = \mathit{tcp} \land \mathit{dport} = 80) \rightarrow \mathit{accept}$
$R_6 : \mathit{deny}$

Fig. 2. Set of rules for our second motivation example.

position of the set of rules as $R_2 : (s \in 111.222.1.0/24 \land d \in 111.222.2.0/24 \land p = \mathit{tcp} \land \mathit{dport} = 80) \rightarrow \mathit{accept}$. Let us also notice that the related rule to the local exception "*The interfaces of firewall FW*$_1$ *are not allowed to access web resources on the Internet*", i.e., the former rule $R_3 : (s \in 111.222.1.1 \land d \in \mathit{any} \land p = \mathit{tcp} \land \mathit{dport} = 80) \rightarrow \mathit{deny}$, is now a global exception, and it must be placed in the first position of the set, i.e., it must be placed as $R_1 : (s \in 111.222.1.1 \land d \in \mathit{any} \land p = \mathit{tcp} \land \mathit{dport} = 80) \rightarrow \mathit{deny}$.

As we can observe, the use of this strategy will continously increase the complexity of the firewall's configuration as the combination of rules will also do. Furthermore, we can even propose combinations of rules that will not be possible to implement by simply ordering the rules. For instance, let us consider the following two condition attributes A and B, such that $A \cap B \neq \emptyset$, and the following two rules: $R_1 : (s \in (A \land \neg B)) \rightarrow \mathit{accept}$; $R_2 : (s \in (B \land \neg A)) \rightarrow \mathit{accept}$. As we have seen in this section, the use of a first matching strategy should easily allow us to separately implement these two rules as follows:

$R_{1,1} : (s \in B) \rightarrow \mathit{deny}$ $R_{1,2} : (s \in A) \rightarrow \mathit{accept}$	$R_{2,1} : (s \in A) \rightarrow \mathit{deny}$ $R_{2,2} : (s \in B) \rightarrow \mathit{accept}$

However, the simple ordering of rules for such an example will not allow us to find out any appropriate combination of rules R_1 and R_2. Instead, we should first compute $A \cap B$ and then transform the previous rules as follows:

$R_{1,1} : (s \in (A \cap B)) \rightarrow \mathit{deny}$ $R_{1,2} : (s \in A) \rightarrow \mathit{accept}$	$R_{2,1} : (s \in (A \cap B)) \rightarrow \mathit{deny}$ $R_{2,2} : (s \in B) \rightarrow \mathit{accept}$

and finally deploy the following set of rules:

$R_1 : (s \in (A \cap B) \rightarrow \mathit{deny}$
$R_2 : (s \in A) \rightarrow \mathit{accept}$
$R_3 : (s \in B) \rightarrow \mathit{accept}$

We can thus conclude that through this strategy the handling of exceptions can lead to very complex configurations and even require additional computations and transformations processes. The administration of the final setup becomes, moreover, an error prone difficult task. Other strategies, like the segmentation of condition attributes or the use of a partial order of rules, will allow us to perform similar managements with better results. We see these other two strategies in the following section.

2.2 Segmentation of Condition Attributes

A second solution when managing exceptions on access control policies is to directly exclude the conditions from the set of rules. In [7, 6], for example, we presented a rewriting mechanism for such a purpose. Through this rewriting mechanism, one may specify the handling of exceptions in the form $R_1 : (s \in (A \wedge \neg B)) \rightarrow accept$ by simply transforming it into the following rule:

$$\boxed{R_1 : (s \in (A - B)) \rightarrow accept}$$

The deployment of our motivation example, i.e., "*All the hosts in (Private ∧ ¬Admin ∧ ¬Interf-fw) are allowed to access web resources on (Internet ∧ ¬Corporate)*", through this new strategy, will be managed as follows. We first obtain the set of exclusions, i.e., (*Private – Admin – Interf-fw*) and (*Internet – Corporate*):

*Private = 111.222.1.**
Admin = [111.222.1.13, 111.222.1.25]
Interf-fw = {111.222.1.1, 111.222.100.1}
  Private – Admin – Interf-fw $\rightarrow$ *[111.222.1.2, 111.222.1.12]* $\cup$ *[111.222.1.26, 111.222.1.254]*
*Internet = *.*.*.**
Corporate = 111.222..**
  Internet – Corporate $\rightarrow$ *[0.0.0.1, 111.222.255.254]* $\cup$ *[111.223.1.1, 255.255.255.254]*

Then, we must deploy the following rules:

$$\begin{array}{l}
R_1 : (s \in [111.222.1.2, 111.222.1.12] \wedge d \in [0.0.0.1, 111.222.255.254] \quad \backslash \\
\qquad \wedge\, p = tcp \wedge dport = 80) \rightarrow accept \\
R_2 : (s \in [111.222.1.26, 111.222.1.255] \wedge d \in [0.0.0.1, 111.222.255.254] \quad \backslash \\
\qquad \wedge\, p = tcp \wedge dport = 80) \rightarrow accept \\
R_3 : (s \in [111.222.1.2, 111.222.1.12] \wedge d \in [111.223.1.1, 255.255.255.254] \quad \backslash \\
\qquad \wedge\, p = tcp \wedge dport = 80) \rightarrow accept \\
R_4 : (s \in [111.222.1.26, 111.222.1.255] \wedge d \in [111.223.1.1, 255.255.255.254] \quad \backslash \\
\qquad \wedge\, p = tcp \wedge dport = 80) \rightarrow accept \\
R_5 : deny
\end{array}$$

The main advantage of this approach, apart from offering a solution for the management of exceptions, is that the ordering of rules is no longer relevant. Hence, one can perform a second transformation in a positive or negative manner: positive, when generating only permissions; and negative, when generating only prohibitions. Positive rewriting can be used in a closed policy whereas negative rewriting can be used in case of an open policy. After this second rewriting, the security officer will have a clear view of the accepted traffic (in the case of positive rewriting) or the rejected traffic (in the case of negative rewriting). However, it also presents some drawbacks. First, it may lead to very complex configuration setups that may even require a post-process of the different segments. Second, it may involve an important increase of the initial number of rules[2]. Nevertheless, such an increase may only degrade the performance of the firewall whether the associated parsing algorithm of the firewall depends on the number of rules. Third, the managing of rule updates through this strategy may also be very complex, since the addition or elimination of new exceptions may require a further segmentation processing of the rules. Some firewall implementations, moreover, are not able to directly manage ranges (e.g., they can require to transform the range $[111.222.1.2, 111.222.1.12]$ into $\{111.222.1.2/31 \cup 111.222.1.4/29 \cup 111.222.1.12/32\}$), and should require the use of third party tools.

2.3 Partial Ordering of Rules

To our knowledge, the most efficient solution to manage the problem of exceptions on access control policies would be by means of a strategy based on partial ordering of rules. Notice that in both first and last matching approaches (cf. Section 2.1), the interpretation of the rules depends on the total order in which the rules are specified, i.e., a total order describes the sequence of rules from a global point of view. However, this ordering of rules can also be implemented in a partial manner, where a set of local sequences of rules are defined for a given specific context.

In the case of NetFilter-based firewalls, for instance, a partial ordering of rules may be achieved through the chain mechanism of IPTables. In this way, we can group sets of rules into different chains, corresponding each one to a given exception. These rules are, moreover, executed in the same order they were included into the chain, i.e., by means of a first match strategy. When a specific traffic matches a rule in the chain, and the decision field of this rule is pointing out to the action $return$, the matching of rules within the given chain stops and the analysis of rules returns to the initial chain. Otherwise, the rest of rules in the chain are considered until a proper match is found. If no rule applies, the default policy of the chain does. Thus, through this new strategy, one may specify the handling of exceptions in the form $R_1 : (s \in (A \wedge \neg B)) \rightarrow accept$ as follows:

$$\boxed{\begin{array}{l} R_1 : (s \in A) \rightarrow jump\_to\ \ chain_A \\ \\ R_2^{chain_A} : (s \in B) \rightarrow return \\ R_1^{chain_A} : accept \end{array}}$$

Regarding the scenario shown in Figure 2, i.e., "*(1) All the hosts in (Private* $\wedge$ *¬Admin* $\wedge$ *¬Interf-fw) are allowed to access web resources on (Internet* $\wedge$ *¬Corporate)*; *(2) All the hosts in (Private* $\wedge$ *¬Interf-fw) are allowed to access web resources on DMZ*", we can now implement such premises via two chains, *private-to-internet* (or *p2i* for short) and *private-to-dmz* (or *p2d* for short), as follows:

[2] This increase is not always a real drawback since the use of a parsing algorithm independent of the number of rules is the best solution for the deployment of firewall technologies [15].

$R_1 : (s \in 111.222.1.0/24 \wedge d \in any \wedge p = tcp \wedge dport = 80) \rightarrow jump\_to\ \ p2i$
$R_2 : (s \in 111.222.1.0/24 \wedge d \in 111.222.2.0/24 \wedge p = tcp \wedge dport = 80) \rightarrow jump\_to\ \ p2d$
$R_3 : deny$

$R_1^{p2i} : (s \in 111.222.1.1) \rightarrow return$
$R_2^{p2i} : (s \in [111.222.1.13, 111.222.1.25]) \rightarrow return$
$R_3^{p2i} : (d \in 111.222.0.0/16) \rightarrow return$
$R_4^{p2i} : accept$

$R_1^{p2d} : (s \in 111.222.1.1) \rightarrow return$
$R_2^{p2d} : accept$

Let us now consider the same rules specified in the syntax of NetFilter. The first two rules create a chain called “private-to-internet” (or *p2i* for short) and a chain called “private-to-dmz” (or *p2d* for short). The third rule corresponds to the positive inclusion condition for the first general case (this way, when a given packet will match this rule, the decision is to jump to the chain *p2i* and check the negative exclusion conditions). Similarly, the fourth rule corresponds to the positive inclusion condition for the second general case. We shall observe that in order to deploy this example over a firewall based on Netfilter we should first verify whether its version of IPTables has been patched to properly manage ranges. We must also correctly define in the final IPTables script those variables such as $PRIVATE, $DMZ, etc.

```
iptables -N p2i
iptables -N p2d

iptables -A FORWARD -s $PRIVATE -p tcp –dport 80 -j p2i
iptables -A FORWARD -s $PRIVATE -d $DMZ -p tcp –dport 80 -j p2d
iptables -A FORWARD -j DROP

iptables -A p2i -s $INTERF_FIREWALL -j RETURN
iptables -A p2i -s $ADMIN -j RETURN
iptables -A p2i -d $CORPORATE -j RETURN
iptables -A p2i -j ACCEPT

iptables -A p2d -s $INTERF_FIREWALL -j DROP
iptables -A p2d -j ACCEPT
```

The main advantages of this strategy (i.e., partial ordering of rules) are threefold. First, it allows a complete separation between exceptions and general rules; second, the ordering of general rules is no longer relevant; and third, the insertion and elimination of both general rules and exception is very simple. We consider, moreover, that a proper reorganization of rules from a total order strategy to a partial order one may also help us to improve not only the handling of exception, but also the firewall’s performance on high-speed networks [16, 12]. In [16], on the one hand, the authors propose a refinement process of rules which generates a decision-like tree implemented through the chain mechanism of IPTables. Their approach basically re-organizes the set of configuration rules into an improved setup, in order to obtain a much flatter design, i.e., a new set of configuration rules, where the number of rules not only decreases but also leads to a more efficient packet matching process. In [12], on the other hand, the authors also propose a reorganization of rules in order to better deploy the final configuration. Nevertheless, both authors in [16] and [12] do not seem to address the handling of exceptions, neither expressiveness aspects of their configuration language – that seems to rely upon partial expressiveness languages.

3 Use of Full Expressiveness

Notice that the solutions above overviewed are always based on partial expressiveness, i.e., they implement security policies by means of security rules whose condition attributes are mainly composed of either positive (e.g., A) or negative (e.g., $\neg A$) statements, but they do not allow us to combine both positive and negative statements (e.g., $A \wedge \neg B$) for a single attribute at the same time. Although we have seen in the previous section that these languages may allow us to specify any possible security policy, they can can lead to very complex configurations when dealing with the management of exceptions. However, the use of both negative and positive statements for each condition attribute may allow us to specify filtering rules in a more efficient way. The use of a structured SQL-like language [9], for example, will allow us to manage the handling of exceptions in the form $R_1 : (s \in (A \wedge \neg\, B)) \rightarrow accept$ through the use of queries like the following ones:

select decision
from $firewall$
where $(s \in A) \wedge (s \notin B)$

select decision **from** $firewall$ **where** $(s \in A)$
minus
select decision **from** $firewall$ **where** $(s \in B)$

However, these kind of languages are not currently being used for the configuration of firewalls or similar devices – at least not for managing exceptions on access control policies, as defined in this paper. We consider that they will allow security officers to deploy the security policies in a more efficient manner, as well as to properly manage the handling of exceptions on access control policies. Let us for example assume that the configuration language we have been using along the examples of this paper allows us the combination of either positive (e.g., A) and negative (e.g., $\neg A$) statements for each attribute of a single filtering rule. For the sake of simplicity, let us just assume the use of a 2-tuple for specifying both positive and negative values of each attribute (e.g., $R_i : (s \in (A \wedge \neg\, B)) \rightarrow accept$ becomes $R_i : (s[+] \in A \wedge s[-] \in B) \rightarrow accept$). Let us also assume that both positive and negative values are initialized to $\emptyset$ by default. Let us finally assume that we rewrite the matching algorithm implemented in our hypothetical firewall FW_1 into Algorithm 1. In this case, we can easily deploy the first motivation example based on Figure 1(b)'s setup, i.e., "*All the hosts in (Private ∧ ¬Admin ∧ ¬Interf-fw) are allowed to access web resources on (Internet ∧ ¬Corporate)*", as follows:

```
Algorithm 1: MatchingAlgorithm
   input : (1) firewall's filtering rules: $r_1 \ldots r_n$;
           (2) firewall's default policy: $policy$;
           (3) packet: $p$
   output: $decision$
1  $decision \leftarrow policy$;
2  $H \leftarrow$ GetPacketHeaders ($p$);
   /* Let $r_i = (A_1^i[+] \in V_1^+) \wedge (A_1^i[-] \in V_1^-) \cdots (A_p^i[+] \in V_p^+) \wedge (A_p^i[-] \in V_p^-) \rightarrow d_i$,  */
   /* where $A_{1..p}^i[+]$ and $A_{1..p}^i[-]$ are, respectively, the set of positive and negative  */
   /* attribute conditions of rule $r_i$; and $V_{1..p}^+$ and $V_{1..p}^-$ are, respectively, the set  */
   /* of positive and negative attribute values of rule $r_i$;  */
3  for $i \leftarrow 1$ to $n$ do
4      if $(H_1 \cap V_1^+ \neq \emptyset) \wedge (H_1 \cap V_1^- = \emptyset) \cdots (H_p \cap V_p^+ \neq \emptyset) \wedge (H_p \cap V_p^- = \emptyset)$ then
5          $decision \leftarrow d_i$;
6          break; /* Leave the loop */
7  return $decision$;
```

$R_1 : (s[+] \in 111.222.1.0/24 \ \wedge \ s[-] \in \{[111.222.1.13, 111.222.1.25] \ \backslash$
$\cup \ 111.222.1.1\} \ \wedge \ d[+] \in any \ \wedge \ d[-] \in 111.222.0.0/16 \ \wedge \ p[+] = tcp \ \backslash$
$\wedge \ dport[+] = 80) \rightarrow accept$
$R_2 : deny;$

Regarding the second motivation example, i.e., "(1) *All the hosts in (Private ∧ ¬Admin ∧ ¬Interf-fw) are allowed to access web resources on (Internet ∧ ¬Corporate)*; (2) *All the hosts in (Private ∧ ¬Interf-fw) are allowed to access web resources on the zone DMZ*", we can now properly specify the resulting set of rules as follows:

$R_1 : (s[+] \in 111.222.1.0/24 \ \wedge \ s[-] \in \{[111.222.1.13, 111.222.1.25] \ \backslash$
$\cup \ 111.222.1.1\} \ \wedge \ d[+] \in any \ \wedge \ d[-] \in 111.222.3.0/24 \ \wedge \ p[+] = tcp \ \backslash$
$\wedge \ dport[+] \in 80) \rightarrow accept$
$R_2 : (s[+] \in 111.222.1.0/24 \ \wedge \ s[-] \in \ 111.222.1.1\} \ \wedge \ d[+] \in 111.222.2.0/24 \ \backslash$
$\wedge \ p[+] = tcp \ \wedge \ dport[+] \in 80) \rightarrow accept$
$R_3 : deny;$

As we can observe, the use of a language based on both positive and negative statements, when specifying the condition attributes of the security rules of a firewall, allows us a more efficient deployment of policies, even using fewer rules. We therefore consider that the little modification we must perform to improve the expressiveness of current firewall configuration languages may allow us to better afford the managing of exceptions on network access control policies. To verify such an assumption, we implemented a proof-of-concept by extending the matching algorithm of IPTables through a Netfilter extension. Due to space limitation, we do not cover in the paper this first proof-of-concept. A report regarding its implementation and performance is provided in the appendix.

4 Related Work

To our knowledge, very little research has been done on the use of full expressiveness languages for the management of firewall configuration as we address in this paper. In [13], for instance, a SQL-like query language for firewalls, called Structured Firewall Query Language is proposed. The authors do not seem to address, however, whether such a language can be used for examining incoming and outgoing traffic, neither to accept nor discard such traffic. The language seems to only be used for the understanding and analysis of firewall's functionality and behavior, rather than be used to perform packet matching or for expressiveness improvement purposes. Similarly, the authors in [14] propose a firewall analysis tool for the management and testing of global firewall policies through a query-like language. However, the expressiveness power of such a language is very limited (just four condition attributes are allowed), and we doubt it may be useful to address our motivation problem.

Some other approaches for the use of formal languages to address the design and creation of firewall rules have been proposed in [5, 8, 3]. However, those approaches aim at specifying and deploying a global security policy through a refinement process that automatically generates the configuration rules of a firewall from a high level language. Thus, the problem of managing exceptions is handled in those works at a high level, rather than a concrete level, and so, the proper configuration once solved the managing issues shall be implemented through one of the strategies already discussed in Section 2. Finally, some proposals for the reorganization of filtering rules have been presented in [16, 12]. However, and as we already pointed out in Section 2, those approaches do not seem to address the handling of exceptions, neither expressiveness aspects of their configuration languages. Their reordering process aim at simply improve the firewall's performance on high speed networks, rather than to offer an easier way to manage the exclusion of condition attributes.

5 Conclusions

In this paper we have studied current strategies in order to manage and deploy policy exceptions when configuring network security components, such as firewalls and filtering routers. As we have discussed, those components are still being configured by security officers in a manual fashion through partial expresssiveness based languages. We have also discussed how the use of these languages can lead to very complex configurations when dealing with exclusions of general rules that should always apply. We finally pointed out to the necessity of full expressiveness for combining both negative and positive conditions on firewall languages in order to improve this management of exceptions on access control policies. As we have seen, the simple modification of a general packet matching algorithm can allow us to perform a more efficient deployment of policies by using almost always fewer rules.

As work in progress, we are actually evaluating the implementation of the strategy presented in this paper over NetFilter-based firewalls. For the moment, we have slightly modified its matching process according to the algorithm shown in Section 3, through the rewriting of a new matching process for IPTables. This first proof-of-concept demonstrates the practicability of our approach. However, we must conduct more experiments to study the real impact on the performance of Netfilter through real scenarios when using our proposal. We plan to address these evaluations and report the results in a forthcoming paper.

References

1. Alfaro, J. G., Cuppens, F., and Cuppens-Boulahia, N. Analysis of Policy Anomalies on Distributed Network Security Setups. In *11th European Symposium On Research In Computer Security (Esorics 2006)*, pp. 496–511, Hamburg, Germany, 2006.
2. Alfaro, J. G., Cuppens, F., and Cuppens-Boulahia, N. Towards Filtering and Alerting Rule Rewriting on Single-Component Policies. In *Intl. Conference on Computer Safety, Reliability, and Security (Safecomp 2006)*, pp. 182–194, Gdansk, Poland, 2006.
3. Alfaro, J. G., Cuppens, F., and Cuppens-Boulahia, N. Aggregating and Deploying Network Access Control Policies. In *Symposium on Frontiers in Availability, Reliability and Security (FARES), 2nd International Conference on Availability, Reliability and Security (ARES 2007)*, Vienna, Austria, 2007.
4. Alfaro, J. G., Cuppens-Boulahia, N., and Cuppens, F. Complete Analysis of Configuration Rules to Guarantee Reliable Network Security Policies In *International Journal of Information Security*, Springer, 7(2):103-122, April 2008.
5. Bartal, Y., Mayer, A., Nissim, K., Wool, A. Firmato: A novel firewall management toolkit ACM Transactions on Computer Systems (TOCS), 22(4):381–420, 2004.
6. Cuppens, F., Cuppens-Boulahia, N., and Alfaro, J. G. Detection and Removal of Firewall Misconfiguration. In *Intl. Conference on Communication, Network and Information Security (CNIS05)*, pp. 154–162, 2005.
7. Cuppens, F., Cuppens-Boulahia, N., and Alfaro, J. G. Misconfiguration Management of Network Security Components. In *7th Intl. Symposium on System and Information Security*, Sao Paulo, Brazil, 2005.
8. Cuppens, F., Cuppens-Boulahia, N., Sans, T. and Miege, A. A formal approach to specify and deploy a network security policy. In *2nd Workshop on Formal Aspects in Security and Trust*, pp. 203–218, 2004.
9. Date, C. J. A guide to the SQL standard. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.
10. Gabillon, A. A formal access control model for XML databases. Lecture notes in computer science, 3674, pp. 86-103, February 2005.
11. Godik, S., Moses, T., and et al. eXtensible Access Control Markup Language (XACML) Version 2. Standard, OASIS. February 2005.
12. Hamed, H. and Al-Shaer, E. On autonomic optimization of firewall policy organization, Journal of High Speed Networks, 15(3):209–227, 2006.
13. Liu, A. X., Gouda, M. G., Ma, H. H., and Ngu, A. H. Firewall Queries. In *Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS-04)*, pp. 197–212, 2004.
14. Mayer, A., Wool, A., Ziskind, E. Fang: A firewall analysis engine. *Security and Privacy Proceedings*, pp. 177–187, 2000.
15. Paul, O., Laurent, M., and Gombault, S. A full bandwidth ATM Firewall. In *Proceedings of the 6th European Symposium on Research in Computer Security (ESORICS 2000)*, pp. 206–221, 2000.

16. Podey, B., Kessler, T., and Melzer, H.D. Network Packet Filter Design and Performance. *Information Networking*, Lecture notes in computer science, 2662, pp. 803–816, 2003.

APPENDIX 1: IPTABLES FEX

We developed our matching strategy as a Netfilter
extension (cf. **http://www.netfilter.org/documentation/HOWTO/netfilter-extensions-HOWTO.html**).

Extending iptables involved two parts:

(1) extension of kernel's code, by writing a new module
(cf. **fex-kernel-src/linux/net/ipv4/netfilter/ipt_fex.c**)

(2) extension of user space's program **iptables**, by writing a
new shared library.

We performed an initial evaluation of our first prototype and experienced that the use of the
new matching strategy decreases the firewall's performance (cf. Figure 1).

The reason for what the performance seems to decrease is related to the evaluation of
those options associated with explicit matches (cf.
http://iptables-tutorial.frozentux.net/chunkyhtml/x2702.html) of Netfilter.

Looking at the function ipt_do_table of netfilter
(cf. **http://lxr.linux.no/source/net/ipv4/netfilter/ip_tables.c#L217**)
we can notice that netfilter analyzes all the rules of the selected table from
a top-down scope where:

(1) it first analyzes generic matches, i.e., it
first analyzes, through function ip_packet_match,
(cf. **http://lxr.linux.no/source/net/ipv4/netfilter/ip_tables.c#L85**),
the following options (cf. **http://iptables-tutorial.frozentux.net/chunkyhtml/c2264.html**):

-p, --protocol
-s, --src, --source
-d, --dst, --destination
-i, --in-interface
-o, --out-interface
-f, --fragment

(2) Then, through the macro IPT_MATCH_ITERATE
(cf. **http://lxr.linux.no/source/include/linux/netfilter_ipv4/ip_tables.h#L229**)
it analyzes the rest of matches, i.e., first implicit matches
(cf. **http://iptables-tutorial.frozentux.net/chunkyhtml/x2436.html**):

TCP matches:
  --sport, --source-port
  --dport, --destination-port
  --tcp-flags
  --syn
  --tcp-option

UDP matches:
  --sport, --source-port
  --dport, --destination-port

ICMP matches:
  --icmp-type

and, whether all the previous matches were successfully
executed, then it analyzes those explicit matches
(cf. **http://iptables-tutorial.frozentux.net/chunkyhtml/x2702.html**),
i.e., matches registered by extended modules through function
ipt_register_match
(cf. **http://lxr.linux.no/source/include/linux/netfilter_ipv4/ip_tables.h#L277**)
and specified from the user space by using the option -m (such our extended match
specified by using -m fex), such as:

-m length
-m limit
-m mac
-m multiport
-m owner
...

The pseudocode for that process should be the following:

```
foreach packet{
 foreach rule in related table{
  foreach match in generic-matches{
   if match() returns 0 then exit();
  }
  foreach match in implicit-explicit-matches{
    if match() returns 0 then exit();
  }
  jump_to_rule_target();
 }
}
```

where:

-- match()
(**http://lxr.linux.no/source/include/linux/netfilter_ipv4/ip_tables.h#L240**)

-- jump_to_rule_target()
(**http://lxr.linux.no/source/net/ipv4/netfilter/ip_tables.c#L271**)

So, to conclude, we consider that the decrease of performance when using our extended match option is due to the internal evaluation process of netfilter, which evaluates each match of each rule in a consecutive manner, i.e., sequentially, rather than the processing time introduced by our extended matching functions.

APPENDIX 2: EVALUATION

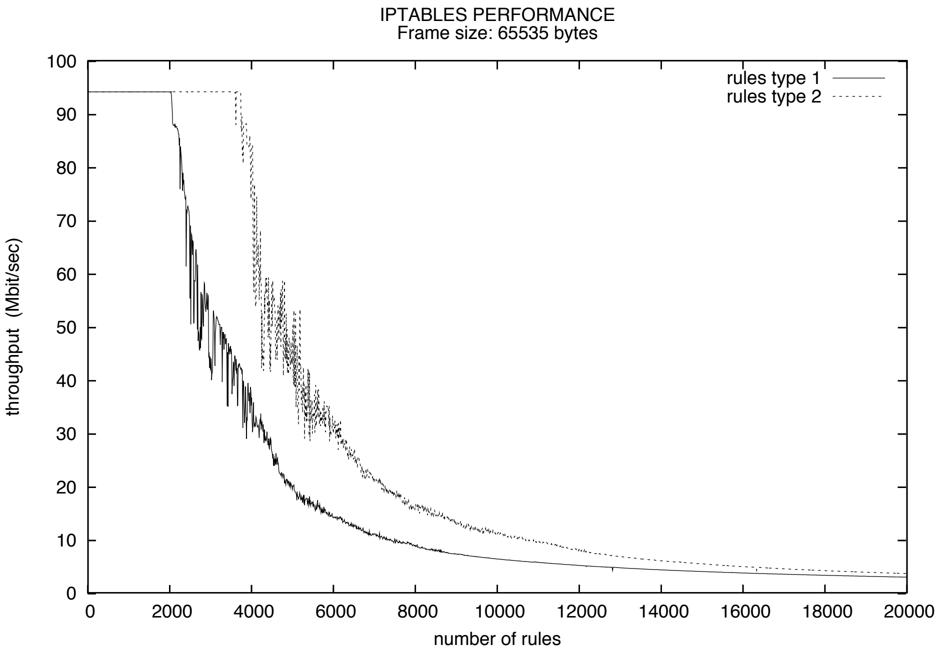


Figure 1: Throughtput evaluation.

– Rules type 2:

```
$>iptables -A OUTPUT -s xxx.xxx.xxx.xxx/xx -d yyy.yyy.yyy.yyy/yy -j DROP
```

– Rules type 1:

```
$>iptables -A OUTPUT ——m extended ——s[+] www.www.www.www-www.www.www.www  \
                                  ——s[-] xxx.xxx.xxx.xxx-xxx.xxx.xxx.xxx  \
                                  ——d[+] yyy.yyy.yyy.yyy-yyy.yyy.yyy.yyy  \
                                  ——d[-] zzz.zzz.zzz.zzz-zzz.zzz.zzz.zzz  \
                                  -j DROP
```

APPENDIX 3: CODE

From: fex-kernel-src/linux/net/ipv4/netfilter/ipt_fex.c

```
#include <linux/module.h>
#include <linux/skbuff.h>
#include <linux/ip.h>
#include <linux/netfilter_ipv4/ip_tables.h>
#include <linux/netfilter_ipv4/ipt_fex.h>


static int
match(const struct sk_buff *skb,
      const struct net_device *in,
      const struct net_device *out,
      const void *matchinfo,
      int offset, int *hotdrop){
        const struct ipt_fex_info *info = matchinfo;
        const struct iphdr *iph = skb->nh.iph;

        if (info->flags & FEX_SRC) {
               if (((ntohl(iph->saddr) < ntohl(info->src.min_ip))
                      || (ntohl(iph->saddr) > ntohl(info->src.max_ip)))){
                      return 0;
               }
        }
        if (info->flags & FEX_DST) {
               if (((ntohl(iph->daddr) < ntohl(info->dst.min_ip))
                      || (ntohl(iph->daddr) > ntohl(info->dst.max_ip)))){
                      return 0;
               }
        }
        if (info->flags & FEX_NSRC) {
               if (((ntohl(iph->saddr) >= ntohl(info->nsrc.min_ip))
                      && (ntohl(iph->saddr) <= ntohl(info->nsrc.max_ip)))){
                      return 0;
               }
        }
        if (info->flags & FEX_NDST) {
               if (((ntohl(iph->daddr) >= ntohl(info->ndst.min_ip))
                      && (ntohl(iph->daddr) <= ntohl(info->ndst.max_ip)))){
                      return 0;
               }
        }
        return 1;
}

static int check(const char *tablename,
                const struct ipt_ip *ip,
                void *matchinfo,
                unsigned int matchsize,
                unsigned int hook_mask){

        /* verify size */
        if (matchsize != IPT_ALIGN(sizeof(struct ipt_fex_info)))
               return 0;
        return 1;
}

static struct ipt_match fex_match ={
        .list = { NULL, NULL },
        .name = "fex",
        .match = &match,
        .checkentry = &check,
        .destroy = NULL,
        .me = THIS_MODULE
};

static int __init init(void){
        return ipt_register_match(&fex_match);
}

static void __exit fini(void){
        ipt_unregister_match(&fex_match);
}

module_init(init);
module_exit(fini);
```

From: fex-kernel-src/linux/include/linux/netfilter_ipv4/ipt_fex.h

```
#ifndef _IPT_FEX_H
#define _IPT_FEX_H

#define FEX_SRC             0x01    /* Match source IP address */
#define FEX_DST             0x02    /* Match destination IP address */
#define FEX_NSRC            0x10    /* Shouldn't match source IP address */
#define FEX_NDST            0x20    /* Shouldn't match destination IP address */

struct ipt_fex {
       /* Inclusive: network order. */
       u_int32_t min_ip, max_ip;
};

struct ipt_fex_info
{
  struct ipt_fex src;
  struct ipt_fex dst;
  struct ipt_fex nsrc;
  struct ipt_fex ndst;

  /* Flags from above */
  u_int8_t flags;
};

#endif /* _IPT_FEX_H */
```

From: fex-kernel-src/linux/net/ipv4/netfilter/ipt_fex.c

```
#include <linux/module.h>
#include <linux/skbuff.h>
#include <linux/ip.h>
#include <linux/netfilter_ipv4/ip_tables.h>
#include <linux/netfilter_ipv4/ipt_fex.h>

MODULE_LICENSE("GPL");

static int
match(const struct sk_buff *skb,
      const struct net_device *in,
      const struct net_device *out,
      const void *matchinfo,
      int offset, int *hotdrop)
{
        const struct ipt_fex_info *info = matchinfo;
        const struct iphdr *iph = skb->nh.iph;

        if (info->flags & FEX_SRC) {
               if (((ntohl(iph->saddr) < ntohl(info->src.min_ip))
                      || (ntohl(iph->saddr) > ntohl(info->src.max_ip)))){
                      return 0;
               }
        }
        if (info->flags & FEX_DST) {
               if (((ntohl(iph->daddr) < ntohl(info->dst.min_ip))
                      || (ntohl(iph->daddr) > ntohl(info->dst.max_ip)))){
                      return 0;
               }
        }
        if (info->flags & FEX_NSRC) {
               if (((ntohl(iph->saddr) >= ntohl(info->nsrc.min_ip))
                      && (ntohl(iph->saddr) <= ntohl(info->nsrc.max_ip)))){
                      return 0;
               }
        }
        if (info->flags & FEX_NDST) {
               if (((ntohl(iph->daddr) >= ntohl(info->ndst.min_ip))
                      && (ntohl(iph->daddr) <= ntohl(info->ndst.max_ip)))){
                      return 0;
               }
        }
        return 1;
}

static int check(const char *tablename,
                const struct ipt_ip *ip,
                void *matchinfo,
                unsigned int matchsize,
                unsigned int hook_mask)
{

        /* verify size */
        if (matchsize != IPT_ALIGN(sizeof(struct ipt_fex_info)))
               return 0;

        return 1;
}

static struct ipt_match fex_match =
{
        .list = { NULL, NULL },
        .name = "fex",
        .match = &match,
        .checkentry = &check,
        .destroy = NULL,
        .me = THIS_MODULE
};

static int __init init(void)
{
        return ipt_register_match(&fex_match);
}

static void __exit fini(void)
{
        ipt_unregister_match(&fex_match);
}

module_init(init);
module_exit(fini);
```

From: fex-libipt_fex-src/extensions/libipt_fex.c

```
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#include <getopt.h>

#include <iptables.h>
#include <linux/netfilter_ipv4/ipt_fex.h>

/* Function which prints out usage message. */
static void
help(void)
{
       printf(
"fex match v%s options:\n"
"--s[+] ip-ip       Match source IP in the specified range\n"
"--d[+] ip-ip       Match destination IP in the specified range\n"
"--s[-] ip-ip       Shouldn't match source IP in the specified range\n"
"--d[-] ip-ip       Shouldn't match destination IP in the specified range\n"
"\n",
IPTABLES_VERSION);
}

static struct option opts[] = {
       { "s[+]", 1, 0, '1' },
       { "d[+]", 1, 0, '2' },
       { "s[-]", 1, 0, '3' },
       { "d[-]", 1, 0, '4' },
       {0}
};

static void
parse_fex(char *arg, struct ipt_fex *range)
{
       char *dash;
       struct in_addr *ip;

       dash = strchr(arg, '-');
       if (dash)
              *dash = '\0';

       ip = dotted_to_addr(arg);
       if (!ip)
              exit_error(PARAMETER_PROBLEM, "fex match: Bad IP address `%s'\n",
                         arg);
       range->min_ip = ip->s_addr;

       if (dash) {
              ip = dotted_to_addr(dash+1);
              if (!ip)
                     exit_error(PARAMETER_PROBLEM, "fex match: Bad IP address `%s'\n",
                                dash+1);
              range->max_ip = ip->s_addr;
       } else
              range->max_ip = range->min_ip;
}

/* Function which parses command options; returns true if it
   ate an option */
static int
parse(int c, char **argv, int invert, unsigned int *flags,
      const struct ipt_entry *entry,
      unsigned int *nfcache,
      struct ipt_entry_match **match)
{
       struct ipt_fex_info *info = (struct ipt_fex_info *)(*match)->data;

       switch (c) {
       case '1':
              if (*flags & FEX_SRC)
                     exit_error(PARAMETER_PROBLEM,
                                "fex match: Only use --s[+] ONCE!");
              *flags |= FEX_SRC;

              info->flags |= FEX_SRC;
              parse_fex(optarg, &info->src);

              break;

       case '2':
              if (*flags & FEX_DST)
                     exit_error(PARAMETER_PROBLEM,
                                "fex match: Only use --d[+] ONCE!");
              *flags |= FEX_DST;
```

```
                info->flags |= FEX_DST;
                parse_fex(optarg, &info->dst);

                break;

        case '3':
                if (*flags & FEX_NSRC)
                        exit_error(PARAMETER_PROBLEM,
                                   "fex match: Only use --s[-] ONCE!");
                *flags |= FEX_NSRC;

                info->flags |= FEX_NSRC;
                parse_fex(optarg, &info->nsrc);

                break;

        case '4':
                if (*flags & FEX_NDST)
                        exit_error(PARAMETER_PROBLEM,
                                   "fex match: Only use --d[-] ONCE!");
                *flags |= FEX_NDST;

                info->flags |= FEX_NDST;
                parse_fex(optarg, &info->ndst);

                break;

        default:
                return 0;
        }
        return 1;
}

/* Final check; must have specified --s[+] or --s[-] or  --d[+] or --d[-] */
static void
final_check(unsigned int flags)
{
        if (!flags)
                exit_error(PARAMETER_PROBLEM,
                           "fex match: You must specify `--s[+]' or `--s[-]' or  `--d[+]' or `--
d[-]'");
}

static void
print_fex(const struct ipt_fex *range)
{
        const unsigned char *byte_min, *byte_max;

        byte_min = (const unsigned char *) &(range->min_ip);
        byte_max = (const unsigned char *) &(range->max_ip);
        printf("%d.%d.%d.%d-%d.%d.%d.%d ",
                byte_min[0], byte_min[1], byte_min[2], byte_min[3],
                byte_max[0], byte_max[1], byte_max[2], byte_max[3]);
}

/* Prints out the info. */
static void
print(const struct ipt_ip *ip,
      const struct ipt_entry_match *match,
      int numeric)
{
        struct ipt_fex_info *info = (struct ipt_fex_info *)match->data;

        if (info->flags & FEX_SRC) {
                printf("s[+]=");
                print_fex(&info->src);
        }
        if (info->flags & FEX_NSRC) {
                printf("s[-]=");
                print_fex(&info->nsrc);
        }
        if (info->flags & FEX_DST) {
                printf("d[+]=");
                print_fex(&info->dst);
        }
        if (info->flags & FEX_NDST) {
                printf("d[-]=");
                print_fex(&info->ndst);
        }
}
```

```
/* Saves the union ipt_info in parsable form to stdout. */
static void
save(const struct ipt_ip *ip, const struct ipt_entry_match *match)
{
        struct ipt_fex_info *info = (struct ipt_fex_info *)match->data;

        if (info->flags & FEX_SRC) {
                printf("--s[+] ");
                print_fex(&info->src);
                if (info->flags & FEX_NSRC)
                        fputc(' ', stdout);
        }
        if (info->flags & FEX_NSRC) {
                printf("--s[-] ");
                print_fex(&info->nsrc);
                if (info->flags & FEX_DST)
                        fputc(' ', stdout);
        }
        if (info->flags & FEX_DST) {
                printf("--d[+] ");
                print_fex(&info->dst);
                if (info->flags & FEX_NDST)
                        fputc(' ', stdout);
        }
        if (info->flags & FEX_NDST) {
                printf("--d[-] ");
                print_fex(&info->ndst);
        }
}

static struct iptables_match fex = {
        .next           = NULL,
        .name           = "fex",
        .version        = IPTABLES_VERSION,
        .size           = IPT_ALIGN(sizeof(struct ipt_fex_info)),
        .userspacesize = IPT_ALIGN(sizeof(struct ipt_fex_info)),
        .help           = &help,
        .parse          = &parse,
        .final_check    = &final_check,
        .print          = &print,
        .save           = &save,
        .extra_opts     = opts
};

void _init(void)
{
        register_match(&fex);
}
```

From: http://lxr.linux.no/source/net/ipv4/netfilter/ip_tables.c#L83

```
/* Returns whether matches rule or not. */
static inline int
ip_packet_match(const struct iphdr *ip,
                const char *indev,
                const char *outdev,
                const struct ipt_ip *ipinfo,
                int isfrag)
{
        size_t i;
        unsigned long ret;

#define FWINV(bool,invflg) ((bool) ^ !!(ipinfo->invflags & invflg))

        if (FWINV((ip->saddr&ipinfo->smsk.s_addr) != ipinfo->src.s_addr,
                  IPT_INV_SRCIP)
            || FWINV((ip->daddr&ipinfo->dmsk.s_addr) != ipinfo->dst.s_addr,
                     IPT_INV_DSTIP)) {
                dprintf("Source or dest mismatch.\n");

                dprintf("SRC: %u.%u.%u.%u. Mask: %u.%u.%u.%u. Target: %u.%u.%u.%u.%s\n",
                        NIPQUAD(ip->saddr),
                        NIPQUAD(ipinfo->smsk.s_addr),
                        NIPQUAD(ipinfo->src.s_addr),
                        ipinfo->invflags & IPT_INV_SRCIP ? " (INV)" : "");
                dprintf("DST: %u.%u.%u.%u Mask: %u.%u.%u.%u Target: %u.%u.%u.%u.%s\n",
                        NIPQUAD(ip->daddr),
                        NIPQUAD(ipinfo->dmsk.s_addr),
                        NIPQUAD(ipinfo->dst.s_addr),
                        ipinfo->invflags & IPT_INV_DSTIP ? " (INV)" : "");
                return 0;
        }

        /* Look for ifname matches; this should unroll nicely. */
        for (i = 0, ret = 0; i < IFNAMSIZ/sizeof(unsigned long); i++) {
                ret |= (((const unsigned long *)indev)[i]
                        ^ ((const unsigned long *)ipinfo->iniface)[i])
                        & ((const unsigned long *)ipinfo->iniface_mask)[i];
        }

        if (FWINV(ret != 0, IPT_INV_VIA_IN)) {
                dprintf("VIA in mismatch (%s vs %s).%s\n",
                        indev, ipinfo->iniface,
                        ipinfo->invflags&IPT_INV_VIA_IN ?" (INV)":"");
                return 0;
        }

        for (i = 0, ret = 0; i < IFNAMSIZ/sizeof(unsigned long); i++) {
                ret |= (((const unsigned long *)outdev)[i]
                        ^ ((const unsigned long *)ipinfo->outiface)[i])
                        & ((const unsigned long *)ipinfo->outiface_mask)[i];
        }

        if (FWINV(ret != 0, IPT_INV_VIA_OUT)) {
                dprintf("VIA out mismatch (%s vs %s).%s\n",
                        outdev, ipinfo->outiface,
                        ipinfo->invflags&IPT_INV_VIA_OUT ?" (INV)":"");
                return 0;
        }

        /* Check specific protocol */
        if (ipinfo->proto
            && FWINV(ip->protocol != ipinfo->proto, IPT_INV_PROTO)) {
                dprintf("Packet protocol %hi does not match %hi.%s\n",
                        ip->protocol, ipinfo->proto,
                        ipinfo->invflags&IPT_INV_PROTO ? " (INV)":"");
                return 0;
        }

        /* If we have a fragment rule but the packet is not a fragment
         * then we return zero */
        if (FWINV((ipinfo->flags&IPT_F_FRAG) && !isfrag, IPT_INV_FRAG)) {
                dprintf("Fragment rule but not fragment.%s\n",
                        ipinfo->invflags & IPT_INV_FRAG ? " (INV)" : "");
                return 0;
        }

        return 1;
}
```

From: http://lxr.linux.no/source/include/linux/netfilter_ipv4/ip_tables.h#L229

```
229 #define IPT_MATCH_ITERATE(e, fn, args...)       \
230 ({                                              \
231         unsigned int __i;                       \
232         int __ret = 0;                          \
233         struct ipt_entry_match *__match;        \
234                                                 \
235         for (__i = sizeof(struct ipt_entry);    \
236              __i < (e)->target_offset;          \
237              __i += __match->u.match_size) {    \
238                 __match = (void *)(e) + __i;    \
239                                                 \

240                 __ret = fn(__match , ## args);  \

241                 if (__ret != 0)                 \
242                         break;                  \
243         }                                       \
244         __ret;                                  \
245 })
```

```
247 /* fn returns 0 to continue iteration */
248 #define IPT_ENTRY_ITERATE(entries, size, fn, args...)           \
249 ({                                                              \
250         unsigned int __i;                                       \
251         int __ret = 0;                                          \
252         struct ipt_entry *__entry;                              \
253                                                                 \
254         for (__i = 0; __i < (size); __i += __entry->next_offset) { \
255                 __entry = (void *)(entries) + __i;              \
256                                                                 \
257                 __ret = fn(__entry , ## args);                  \
258                 if (__ret != 0)                                 \
259                         break;                                  \
260         }                                                       \
261         __ret;                                                  \
262 })

277 #define ipt_register_match(mtch)        \
278 ({      (mtch)->family = AF_INET;       \
279         xt_register_match(mtch); })
280 #define ipt_unregister_match(mtch) xt_unregister_match(mtch)
```

From: http://lxr.linux.no/source/net/ipv4/netfilter/ip_tables.c#L215

```
215 /* Returns one of the generic firewall policies, like NF_ACCEPT. */
216 unsigned int
217 ipt_do_table(struct sk_buff **pskb,
218              unsigned int hook,
219              const struct net_device *in,
220              const struct net_device *out,
221              struct ipt_table *table,
222              void *userdata)
223 {
224         static const char nulldevname[IFNAMSIZ] __attribute__((aligned(sizeof(long))));
225         u_int16_t offset;
226         struct iphdr *ip;
227         u_int16_t datalen;
228         int hotdrop = 0;
229         /* Initializing verdict to NF_DROP keeps gcc happy. */
230         unsigned int verdict = NF_DROP;
231         const char *indev, *outdev;
232         void *table_base;
233         struct ipt_entry *e, *back;
234         struct xt_table_info *private;
235 
236         /* Initialization */
237         ip = (*pskb)->nh.iph;
238         datalen = (*pskb)->len - ip->ihl * 4;
239         indev = in ? in->name : nulldevname;
240         outdev = out ? out->name : nulldevname;
241         /* We handle fragments by dealing with the first fragment as
242          * if it was a normal packet.  All other fragments are treated
243          * normally, except that they will NEVER match rules that ask
244          * things we don't know, ie. tcp syn flag or ports).  If the
245          * rule is also a fragment-specific rule, non-fragments won't
246          * match it. */
247         offset = ntohs(ip->frag_off) & IP_OFFSET;
248 
249         read_lock_bh(&table->lock);
250         IP_NF_ASSERT(table->valid_hooks & (1 << hook));
251         private = table->private;
252         table_base = (void *)private->entries[smp_processor_id()];
253         e = get_entry(table_base, private->hook_entry[hook]);
254 
255         /* For return from builtin chain */
256         back = get_entry(table_base, private->underflow[hook]);
257 
258         do {
259                 IP_NF_ASSERT(e);
260                 IP_NF_ASSERT(back);
261                 if (ip_packet_match(ip, indev, outdev, &e->ip, offset)) {
262                         struct ipt_entry_target *t;
263 
264                         if (IPT_MATCH_ITERATE(e, do_match,
265                                               *pskb, in, out,
266                                               offset, &hotdrop) != 0)
267                                 goto no_match;
268 
269                         ADD_COUNTER(e->counters, ntohs(ip->tot_len), 1);
270 

271                         t = ipt_get_target(e);

272                         IP_NF_ASSERT(t->u.kernel.target);
273                         /* Standard target? */
274                         if (!t->u.kernel.target->target) {
275                                 int v;
276 
277                                 v = ((struct ipt_standard_target *)t)->verdict;
278                                 if (v < 0) {
279                                         /* Pop from stack? */
280                                         if (v != IPT_RETURN) {
281                                                 verdict = (unsigned)(-v) - 1;
282                                                 break;
283                                         }
284                                         e = back;
285                                         back = get_entry(table_base,
286                                                          back->comefrom);
287                                         continue;
288                                 }
```

```
                                if (table_base + v != (void *)e + e->next_offset
                                    && !(e->ip.flags & IPT_F_GOTO)) {
                                        /* Save old back ptr in next entry */
                                        struct ipt_entry *next
                                                = (void *)e + e->next_offset;
                                        next->comefrom
                                               = (void *)back - table_base;
                                        /* set back pointer to next entry */
                                        back = next;
                                }

                                e = get_entry(table_base, v);
                        } else {
                                /* Targets which reenter must return
                                   abs. verdicts */
#ifdef CONFIG_NETFILTER_DEBUG
                                ((struct ipt_entry *)table_base)->comefrom
                                        = 0xeeeeeeec;
#endif
                                verdict = t->u.kernel.target->target(pskb,
                                                                     in, out,
                                                                     hook,
                                                                     t->u.kernel.target,
                                                                     t->data,
                                                                     userdata);

#ifdef CONFIG_NETFILTER_DEBUG
                                if (((struct ipt_entry *)table_base)->comefrom
                                    != 0xeeeeeeec
                                    && verdict == IPT_CONTINUE) {
                                        printk("Target %s reentered!\n",
                                               t->u.kernel.target->name);
                                        verdict = NF_DROP;
                                }
                                ((struct ipt_entry *)table_base)->comefrom
                                        = 0x57acc001;
#endif
                                /* Target might have changed stuff. */
                                ip = (*pskb)->nh.iph;
                                datalen = (*pskb)->len - ip->ihl * 4;

                                if (verdict == IPT_CONTINUE)
                                        e = (void *)e + e->next_offset;
                                else
                                        /* Verdict */
                                        break;
                        }
                } else {

                no_match:
                        e = (void *)e + e->next_offset;
                }
        } while (!hotdrop);

        read_unlock_bh(&table->lock);

#ifdef DEBUG_ALLOW_ALL
        return NF_ACCEPT;
#else
        if (hotdrop)
                return NF_DROP;
        else return verdict;
#endif
}
```