Formal Correctness of Conflict Detection for Firewalls

Venanzio Capretta[*]
venanzio@cs.ru.nl

Bernard Stepien
bernard@site.uOttawa.ca

Amy Felty
afelty@site.uOttawa.ca

Stan Matwin[†]
stan@site.uOttawa.ca

School of Information Technology and Engineering (SITE)
University of Ottawa, Canada

ABSTRACT

We describe the formalization of a correctness proof for a conflict detection algorithm for firewalls in the Coq Proof Assistant. First, we give formal definitions in Coq of a firewall access *rule* and of an access *request* to a firewall. Formally, two rules are in conflict if there exists a request on which one rule would allow access and the other would deny it. We express our algorithm in Coq, and prove that it finds all conflicts in a set of rules. We obtain an OCaml version of the algorithm by direct program extraction. The extracted program has successfully been applied to firewall specifications with over 200,000 rules.

Categories and Subject Descriptors

F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification; C.2.0 [**General**]: Security and protection

General Terms

Security, Verification

Keywords

Coq, Firewall

1. INTRODUCTION

Firewalls are an essential component of enterprise security. Several factors with respect to current use and configuration of firewalls, however, can lead to security compromises. The distributed nature of firewalls is one such factor. Firewalls operate in an environment of routers that are used to interface with the outside world. They also partition an internal network of an enterprise in order to reduce security risks by restricting the access to machines strictly to the authorized users of these machines. It is thus essential to at least attempt to reconcile the intentions of various interfaces so as to avoid conflicting policies.

Another factor that can lead to security problems is the low level of firewall specification languages and the primitive user interfaces that are used in order to program firewalls. Currently, most users prefer to download a rule base to their computers so as to modify it using a conventional text editor. This is a fairly primitive solution to ensuring a correct firewall since a text editor has no specialized features for checking the syntax or semantics for the particular rule description language used. In addition, as rule sets become larger and more complex, modifying them can result in unanticipated changes to the overall policy.

Factors such as those described above have been identified over the last several years and a number of tools have been developed to cope with various aspects of the problem. A classification of the kinds of anomalies in firewall configurations is described in several papers, e.g. [1, 15, 32]. In this paper, we address the problem of conflicts in firewall rules. In particular, we detect pairs of rules such that, given a specific access request, one rule permits the access and the other denies it.

We consider the Cisco [7] firewall specification language in particular. This language has several code-saving features that enable a user to write a single rule for several source or destination hosts. Such techniques often involve using value masks, port number ranges, or lists of port number specifications for which a particular rule is applicable. Thus each rule applies to a possibly wide range of access requests, and it is often the case that more than one rule will apply to any particular request. In practice, only the first rule that is encountered from the top of the rule base is executed, thus ignoring any further rules pertaining to the same targets. In some cases, a firewall administrator may intentionally introduce certain kinds of overlaps knowing that only the first rule is important: a default policy is specified by a later *blanket* rule, which can be contradicted on smaller ranges by earlier rules. In this case the conflict is intended. However, the specification language does not explicitly mark blanket rules, so there is no way to distinguish these false conflicts from actual mistakes in the firewall.

[*]now at Computer Science Institute (iCIS), Radboud University Nijmegen, The Netherlands

[†]also affiliated with Institute for Computer Science, Polish Academy of Sciences, Warsaw, Poland

Moreover, overlaps are likely to occur if more than one person maintains a firewall, as is often the case in a large organization. However, it is quite common that when more than one rule applies, there is an unintended redundancy or conflict—redundancy in the case when all the applicable rules permit the request or all deny the request, and conflict when at least one applicable rule permits the request and one denies it.

To address the conflict problem, we develop a conflict detection program and prove its correctness in the proof assistant Coq [29, 6]. In the process, we also take care to write an efficient program, so that we may extract it to OCaml and run it on actual firewall specifications. The extracted program, in fact, can detect conflicts in firewalls with hundreds of thousands of rules.

A variety of tools have been designed to address specific problems in firewall specifications. They can roughly be divided into tools that allow querying firewalls to get various kinds of information as well as high-level help with debugging, and tools which perform analyses such as conflict and redundancy checking. A good overview can be found in the related work section of [32]. We discuss some of them in more detail later. None that we are aware of have been formally verified. For example, the work of Eronen and Zitting [11] addresses the conflict problem directly. They implement an expert system in Prolog for this task. Although they discuss conflicts, they do not define them formally. In fact, by formalizing our results, we were led to a more general definition of conflict than the one used by their system.

The advantages and contributions of this work can be summarized as follows:

- *Uses existing firewall specification language:* We provide a tool that can work directly on firewall configurations expressed in languages such as the Cisco firewall specification language, to make them more secure. Other tools described in the literature, as far as we know, which work directly with languages used by practitioners, do not scale up as much as our approach.

- *Can be adopted directly by a firewall administrator:* Our tool provides a powerful aid to firewall administrators, which they can use directly without being required to change the way firewalls are currently programmed. It does not modify a firewall specification, but reports potential conflicts to a firewall administrator, who decides if the reported conflict is an intended one, or needs to be addressed.

- *Handles ranges and masks in full generality:* Most algorithms and tools simplify the kinds of ranges that are allowed when specifying a set of hosts or ports. Although it is usually better "programming practice" to use single intervals when writing rules, the fact that firewall configuration languages allow more than that means that any tool that claims to do a full analysis must handle all that the language is capable of expressing. We did not find any other work that discussed handling this level of generality.

- *Formally verified:* The proof itself is not complicated, but with formal verification as an initial goal, we were led to an elegant treatment of ranges, masks, and intersections that led to a simplified algorithm, and a verification that took some effort but was well within the capabilities of existing verification tools.

- *Efficient extracted program:* As well as a program that can be executed directly in Coq, we also were able to extract an efficient OCaml program whose correctness is guaranteed. The extracted program can handle large sets of rules, with tens to hundreds of thousands of rules.

- *Scalability:* A variety of other tools discussed later also scale up well, but all those that we found that do so also simplify the rule format in some way to abstract away the complications found in real firewall configuration languages such as the one considered here.

The files of the Coq formalization and the extracted OCaml program are available at: `http://www.site.uottawa.ca/~afelty/coq/fmse07_coq.html`

2. RULES AND REQUESTS

A firewall specification is a sequence of rules stating access permissions to some resources. Cisco firewall rules have several formats depending on the purpose of the rule and also on the version of the Cisco specification language. We present some examples and then show how we represent rules in general in Coq.

A rule consists of a group number, an action (permit or deny), a specification of a set of source hosts, a set of destination hosts, and an interval of port numbers to which this rule applies. In the following rule, group 105 denies UDP access to port numbers greater than 100 for traffic originating at host IP address 10.0.0.2 and terminating at host 10.0.1.2.

```
access-list 105 deny udp
            host 10.0.0.2 host 10.0.1.2
            gt 100
```

The two `host` keywords indicate a single source host, and a single destination host, respectively. The expression `gt 100` indicates that the rule applies to port numbers in the interval `101` to the maximum possible port number. Note that this rule specifies only one port; our algorithm will handle the general case where both source and destination ports can be included.

The following rule illustrates two ways to indicate a range of possible hosts.

```
access-list 153 permit udp
            any 10.0.0.0 0.0.0.255
            eq 124
```

The keyword `any` is used to indicate that the rule is applicable to source addresses in the entire allowable range of values. This is followed by a specification of a range of destination IP addresses. In general, a range is specified by a base IP number followed by a mask that is to be interpreted in its binary representation. When a bit in the mask is set to 0, this indicates that the corresponding bit number in the address must match the corresponding bit in the base. Otherwise a 1 bit has the effect of a wild card on that same bit number. In the above example, `10.0.0.0` is the base IP address and `0.0.0.255` the mask. Thus, the destination consists of all IP addresses in the range from `10.0.0.0` to `10.0.0.255`.

A firewall is a program that takes access requests as input and gives a decision, *permit* or *deny*, as output according to the set of firewall rules. A request includes a specific communication protocol, source address, source port number, destination address, and destination port number. We formalize requests as records with fields implemented in Coq as follows:

```
Record request : Set := ask_request
  { req_protocol : protocol;
    req_source_ip : ip_address;
    req_source_pn : port_number;
    req_dest_ip : ip_address;
    req_dest_pn : port_number }.
```

where protocol, ip_address, and port_number are appropriate Coq representations of those notions. For example, the rule formally written as

ask_request tcp (ip 1 0 0 1) 19 (ip 1 5 0 100) 35

is read as: *Request to transmit from the computer with IP address 1.0.0.1, via port number 19, to the computer with IP address 1.5.0.100, via port number 35, using the TCP protocol.*

A firewall rule is also formalized as a Coq record, where the fields for IP addresses allow ranges, and the fields for port numbers allow intervals:

```
Record access_rule : Set := rule
  { r_action : action;
    r_protocol : protocol;
    r_source_ip : ip_range;
    r_source_pn : pn_interval;
    r_dest_ip : ip_range;
    r_dest_pn : pn_interval }.
```

A *rule set*, which defines a firewall, is just a list of rules:

Definition rule_set := list access_rule.

Notation: lists in Coq have the constructors nil for the empty list and _::_ for the cons operator, so a rule set has the form: $r_0::r_1::r_2::\cdots::r_n::$nil, where $r_1,\ldots,r_n$ are access rules.

We have developed a parser for Cisco IOS extended rules as defined in [26]. This parser is a multi target language tool that produces representations in Prolog and the various Coq representations that we have been experimenting with during this research. It handles, among other things, the range keywords that enable the specification of ranges of IP addresses and port numbers. Each occurrence of these keywords is translated into the appropriate Coq range: the keyword `any` that matches all IP addresses, is represented by an IP range with a mask containing all 1s; the specification `gt 100` is represented as the range $(101, 65535)$, and the equality specification `eq 124` as the range $(124, 124)$. The two IOS examples rules above are translated as follows:

```
(rule  deny udp
       (ip_range (ip 10 0 0 2) (ip 0 0 0 0)) (101, 65535)
       (ip_range (ip 10 0 1 2) (ip 0 0 0 0)) (101, 65535))::
(rule  permit udp
       (ip_range (ip 0 0 0 0) (ip 255 255 255 255)) (124, 124)
       (ip_range (ip 10 0 0 0) (ip 0 0 0 255)) (124, 124))::nil.
```

When a request is submitted, the firewall checks it against all the rules of the rule set and produces the appropriate action. A rule applies to a request if:

- req_protocol matches r_protocol;
- req_source_ip is in range of r_source_ip;
- req_source_pn is in range of r_source_pn;
- req_dest_ip is in range of r_dest_ip;
- req_dest_pn is in range of r_dest_pn.

If they all match, the decision given by r_action (permit or deny) is taken. If some of the fields don't match, the rule does not apply.

The question of correctness arises because a request may match several different rules. If they specify inconsistent actions, we say that there is a *conflict* in the rule set.

3. SPECIFICATION OF CORRECTNESS

The conflict detection software should be a program with type:

$$\text{find\_conflicts} : \text{rule\_set} \to \text{list}\ (\mathbb{N} \times \mathbb{N}).$$

Given a rule set as input, it outputs the list of indices (positions in the rule set) of those pairs of rules that may generate a conflict. Correctness of the software means that it finds all and only the conflicting rules.

We start the specification by a formal definition of conflict. First of all we use two relations between access rules and requests, specifying when the request is permitted or denied by the given rule:

$$\begin{aligned}&\text{rule\_permit} : \text{access\_rule} \to \text{request} \to \text{Prop};\\ &\text{rule\_deny} : \text{access\_rule} \to \text{request} \to \text{Prop}.\end{aligned}$$

Given an access rule r and a request q, (rule_permit r q) is a proposition: it is true if q is permitted by r; it is false if q is either denied by or does not match r. Similarly, (rule_deny r q) is true if and only if r denies q. The two relations are both false in case q does not match r.

We say that two rules are in conflict if there is some request on which they give opposite actions:

$$\begin{aligned}&\text{rule\_conflict} : \text{access\_rule} \to \text{access\_rule} \to \text{Prop}\\ &\text{rule\_conflict}\ q_1\ q_2 :=\\ &\quad \exists r : \text{request},\ (\text{rule\_permit}\ q_1\ r \wedge \text{rule\_deny}\ q_2\ r) \vee\\ &\qquad\qquad\qquad (\text{rule\_deny}\ q_1\ r \wedge \text{rule\_permit}\ q_2\ r).\end{aligned}$$

Finally, we define a three-place relation (rs_conflict $\bar{r}$ i j), where $\bar{r}$ is a rule set and i and j are indices, that holds if the ith and jth rules of $\bar{r}$ conflict:

$$\begin{aligned}&\text{rs\_conflict} : \text{rule\_set} \to \mathbb{N} \to \mathbb{N} \to \text{Prop}\\ &\text{rs\_conflict}\ \bar{r}\ i\ j :=\\ &\quad \{H_i : i < (\text{length}\ \bar{r}) \wedge\\ &\quad \{H_j : j < (\text{length}\ \bar{r}) \wedge\\ &\qquad (\text{rule\_conflict}\ (\text{Nth}\ \bar{r}\ i\ H_i)\ (\text{Nth}\ \bar{r}\ j\ H_j))\}\}.\end{aligned}$$

This reads: *Both i and j are smaller than the length of the rule set $\bar{r}$ and there is a conflict between rules number i and j of $\bar{r}$.* Two special notations have been used: the notation (Nth $\bar{r}$ i H_i) denotes the ith element of the list $\bar{r}$, requiring a proof H_i that i is strictly smaller than the length of $\bar{r}$; the notation $\{(h : P) \wedge Q[h]\}$ denotes *dependent conjunction* [22], that is, the conjunction of two propositions P and Q where Q depends on the proof h of P. Such dependency of functions and propositions on proofs is quite common in intensional type theory, on which Coq is based: it is a direct consequence of the Curry-Howard isomorphism [17, 27], which identifies propositions with data-types and proofs with programs.

The correctness specification states that the conflict detection program find_conflicts finds all and only such conflicts. It splits into two properties: soundness and completeness.

$$\begin{array}{l}\mathsf{conflict\_soundness} : \\ \quad \forall(\bar{r} : \mathsf{rule\_set})(i\ j : \mathbb{N}), \\ \quad (i, j) \in (\mathsf{find\_conflicts}\ \bar{r}) \to \mathsf{rs\_conflict}\ \bar{r}\ i\ j.\end{array}$$

$$\begin{array}{l}\mathsf{conflict\_completeness} : \\ \quad \forall(\bar{r} : \mathsf{rule\_set})(i\ j : \mathbb{N}), \\ \quad i < j \to \mathsf{rs\_conflict}\ \bar{r}\ i\ j \to (i, j) \in (\mathsf{find\_conflicts}\ \bar{r}).\end{array}$$

In the statement of completeness, we require $i < j$ for efficiency. We want to detect every pair of conflicting rules only once (and a rule never conflicts with itself).

4. CONFLICT DETECTION

Before programming the conflict detection algorithm and proving its soundness and completeness, let us be more specific about the definition of the fields in a request and the corresponding ranges in a rule. For every field, we define a *membership predicate* between values and ranges that characterizes the values in the range. Towards the construction of a conflict detection program, we also define an *intersection check* boolean function of two ranges that takes the value true if and only if there exists a value that belongs to both ranges.

Actions and protocols are simple sets. Actions have just two discrete values. For simplicity, our definition of protocol also has just two values, but could easily be generalized to include other communication protocols.

$$\begin{array}{l}\mathsf{action} : \mathsf{Set} := \mathsf{permit} \mid \mathsf{deny}; \\ \mathsf{protocol} : \mathsf{Set} := \mathsf{udp} \mid \mathsf{tcp}.\end{array}$$

Here we do not have ranges but single values in both requests and rules, therefore we do not need a range membership relation (equality will do) and the boolean check function is just an identity check:

$$\begin{array}{l}\mathsf{action\_bool} : \mathsf{action} \to \mathsf{action} \to \mathbb{B}; \\ \mathsf{protocol\_bool} : \mathsf{protocol} \to \mathsf{protocol} \to \mathbb{B}.\end{array}$$

Port numbers are integers and the rules prescribe an integer interval for them, specified by its extremities. The membership relation and the intersection check function are straightforward (we use the operators $\sqcap$, $\sqcup$, $\doteq$, and $\sqsubseteq$ to denote the boolean versions of conjunction, disjunction, equality, and order):

$$\begin{array}{l}\mathsf{port\_number} : \mathsf{Set} := \mathbb{Z}; \\ \mathsf{pn\_interval} : \mathsf{Set} := \mathsf{port\_number} \times \mathsf{port\_number}; \\ \\ \mathsf{pn\_match} : \mathsf{port\_number} \to \mathsf{pn\_interval} \to \mathsf{Prop} \\ \mathsf{pn\_match}\ n\ (i_1, i_2) := i_1 \leq n \leq i_2; \\ \\ \mathsf{pn\_bool} : \mathsf{pn\_interval} \to \mathsf{pn\_interval} \to \mathbb{B} \\ \mathsf{pn\_bool}\ (x_1, y_1)\ (x_2, y_2) := \ ((x_1 \sqsubseteq y_1) \sqcap (x_2 \sqsubseteq y_2)) \sqcap \\ \qquad\qquad\qquad\qquad\qquad\quad ((x_1 \sqsubseteq y_2) \sqcap (x_2 \sqsubseteq y_1)).\end{array}$$

In the definition of pn_bool, the first line checks that both intervals are non-empty, the second that they intersect.

The situation for IP addresses is a bit more complicated: An IP address is specified by four bytes of eight bits each. In the Coq definitions, the type bit is just $\mathbb{B}$, byte is bit $\times$ bit $\times$ bit $\times$ bit $\times$ bit $\times$ bit $\times$ bit $\times$ bit, and ip_address is byte $\times$ byte $\times$ byte $\times$ byte. To facilitate the writing of specific IP addresses, the already mentioned operation ip takes four integers and converts them into a four-byte address. As illustrated in the previous section, ranges are not simple intervals as in the case of port numbers but are given by a *base address* and a *mask*. The mask specifies which bits of the base address are to be considered variable and which ones are to be considered fixed. We explain this further here with a more detailed example, which illustrates our notation and motivates our definitions. If the base address is (*ip* 140 101 171 31) and the mask is (*ip* 24 7 56 255), then, writing them in binary form in the first two lines, we get the matching pattern in the third line of the following scheme:

```
base:    10001100.01100101.10101011.00011111
mask:    00011000.00000111.00111000.11111111
pattern: 100**100.01100***.10***011.********
```

This shows that the matching addresses must have the same bit values as the base in the positions where the mask has a 0, and can have any value in the positions where the mask has a 1. In practice this system is used to define simple intervals by having a mask with all 0s in the higher positions and all 1s in the lower positions. One of the common errors in defining access rules is to give an incorrect mask, thus specifying unintended addresses. The formal definition of an IP address range is a record with two fields:

$$\begin{array}{l}\textbf{Record}\ \mathsf{ip\_range} : \mathsf{Set} := \mathsf{mk\_range} \\ \quad \{\ \mathsf{ip\_base} : \mathsf{ip\_address}; \\ \quad\quad \mathsf{ip\_mask} : \mathsf{ip\_address}\ \}.\end{array}$$

The matching relation between addresses and ranges is defined first for bits and bytes, with an extra argument for the mask, and then for the full IP address:

$$\begin{array}{l}\mathsf{bit\_match} : \mathsf{bit} \to \mathsf{bit} \to \mathsf{bit} \to \mathsf{Prop} \\ \mathsf{bit\_match}\ b_1\ b_2\ b_m := (b_m = \mathsf{true}) \vee (b_1 = b_2); \\ \\ \mathsf{byte\_match} : \mathsf{byte} \to \mathsf{byte} \to \mathsf{byte} \to \mathsf{Prop} \\ \mathsf{byte\_match}\ (b_{11}, \ldots, b_{18})\ (b_{21}, \ldots, b_{28})\ (b_{m1}, \ldots, b_{m8}) \\ \quad := \mathsf{bit\_match}\ b_{11}\ b_{21}\ b_{m1} \wedge \ldots \wedge \mathsf{bit\_match}\ b_{18}\ b_{28}\ b_{m8}; \\ \\ \mathsf{ip\_match} : \mathsf{ip\_address} \to \mathsf{ip\_range} \to \mathsf{Prop} \\ \mathsf{ip\_match}\ (a_1, \ldots, a_4)\ (\mathsf{mk\_range}\ (b_1, \ldots, b_4)\ (m_1, \ldots, m_4)) \\ \quad := \mathsf{byte\_match}\ a_1\ b_1\ m_1 \wedge \ldots \wedge \mathsf{byte\_match}\ a_4\ b_4\ m_4.\end{array}$$

Consequently, two IP address ranges overlap if in every bit position one of the following two conditions holds: at least one of the two masks has a 1 or the two base bits are the same. This leads to the following definitions:

$$\begin{array}{l}
\mathsf{bit\_mask\_bool} : \mathsf{bit} \to \mathsf{bit} \to \mathsf{bit} \to \mathsf{bit} \to \mathbb{B} \\
\mathsf{bit\_mask\_bool}\ b_1\ m_1\ b_2\ m_2 := m_1 \sqcup m_2 \sqcup (b_1 \doteq b_2); \\
\\
\mathsf{byte\_mask\_bool} : \mathsf{byte} \to \mathsf{byte} \to \mathsf{byte} \to \mathsf{byte} \to \mathbb{B} \\
\mathsf{byte\_mask\_bool}\ \ (b_{11}, \ldots, b_{18})\ (m_{11}, \ldots, m_{18}) \\
\qquad\qquad\qquad\quad (b_{21}, \ldots, b_{28})\ (m_{21}, \ldots, m_{28}) \\
\quad := \ (\mathsf{bit\_mask\_bool}\ b_{11}\ m_{11}\ b_{21}\ m_{21}) \sqcap \ldots \\
\qquad\quad \ldots \sqcap (\mathsf{bit\_mask\_bool}\ b_{18}\ m_{18}\ b_{28}\ m_{28}); \\
\\
\mathsf{ip\_address\_bool} : \mathsf{ip\_range} \to \mathsf{ip\_range} \to \mathbb{B} \\
\mathsf{ip\_address\_bool}\ \ (\mathsf{mk\_range}\ (b_{11},\ \ldots, b_{14})\ (m_{11},\ \ldots, m_{14})) \\
\qquad\qquad\qquad\quad (\mathsf{mk\_range}\ (b_{21},\ \ldots, b_{24})\ (m_{21},\ \ldots, m_{24})) \\
\quad := \ (\mathsf{byte\_mask\_bool}\ b_{11}\ m_{11}\ b_{21}\ m_{21}) \sqcap \ldots \\
\qquad\quad \ldots \sqcap (\mathsf{byte\_mask\_bool}\ b_{14}\ m_{14}\ b_{24}\ m_{24}).
\end{array}$$

Having defined the functions that check intersections of ranges for each field, we can finally program a function that compares two rules, verifies if they overlap, and in that case checks whether they give the same action:

$$\begin{array}{l}
\mathsf{conflict\_check} : \mathsf{access\_rule} \to \mathsf{access\_rule} \to \mathbb{B} \\
\mathsf{conflict\_check}\ \ (\mathsf{rule}\ a_1\ p_1\ r_{s1}\ n_{s1}\ r_{d1}\ n_{d1}) \\
\qquad\qquad\qquad\ \ (\mathsf{rule}\ a_2\ p_2\ r_{s2}\ n_{s2}\ r_{d2}\ n_{d2}) \\
\ := \ \mathbf{if}\ (\mathsf{action\_bool}\ a_1\ a_2) \\
\qquad \mathbf{then}\ \mathsf{false} \\
\qquad \mathbf{else}\ \ (\mathsf{protocol\_bool}\ p_1\ p_2) \sqcap \\
\qquad\qquad\quad (\mathsf{ip\_address\_bool}\ r_{s1}\ r_{s2}) \sqcap (\mathsf{pn\_bool}\ n_{s1}\ n_{s2}) \sqcap \\
\qquad\qquad\quad (\mathsf{ip\_address\_bool}\ r_{d1}\ r_{d2}) \sqcap (\mathsf{pn\_bool}\ n_{d1}\ n_{d2}).
\end{array}$$

This states that if the action given by the two rules is the same, then there is no conflict; if the two actions are different, then there is a conflict if all the corresponding fields have non-empty intersection. A slight modification of our algorithm can be used to detect *redundant* rules: rules that apply to the same requests and give the same answer. It is enough to check for non-empty intersections when the action is the same. This redundancy-detection algorithm is very similar to the conflict detection one, so we do not explicitly formulate it.

Finally, the conflict detection algorithm must check all the pairs of rules in the rule set and apply conflict_check to all:

$$\mathsf{find\_conflicts} : \mathsf{rule\_set} \to \mathsf{list}\ (\mathbb{N} \times \mathbb{N}).$$

Pairs of rules are compared in their occurrence order: a rule is compared only with the rules following it, to avoid duplicate comparison.

To implement the procedure efficiently, we define a couple of auxiliary functions to perform careful bookkeeping of the rule indices. The first function, rrs_conflicts, checks a single rule r against a list of rules $\vec{s}$ and returns a list of pairs of indices. We give as inputs also a pair of indices: i is the index of r in the original rule set and n is the index of the first element of $\vec{s}$ in the original rule set. The function is defined by recursion on the structure of $\vec{s}$.

$$\begin{array}{l}
\mathsf{rrs\_conflicts} :\ \ \mathbb{N} \to \mathbb{N} \to \\
\qquad\qquad\qquad\quad \mathsf{access\_rule} \to \mathsf{rule\_set} \to \mathsf{list}\ (\mathbb{N} \times \mathbb{N}) \\
\mathsf{rrs\_conflicts}\ i\ n\ r\ \mathsf{nil} := \mathsf{nil} \\
\mathsf{rrs\_conflicts}\ i\ n\ r\ (h{::}\vec{s}) \\
\quad := \ \mathbf{if}\ (\mathsf{conflict\_check}\ r\ h) \\
\qquad\quad \mathbf{then}\ (i, n){::}(\mathsf{rrs\_conflicts}\ i\ (n+1)\ r\ \vec{s}) \\
\qquad\quad \mathbf{else}\ (\mathsf{rrs\_conflicts}\ i\ (n+1)\ r\ \vec{s}).
\end{array}$$

The second function searches for conflicts inside a tail fragment $\vec{s}$ of the original rule set, using an extra input indicating at what position of the original rule set $\vec{s}$ starts:

$$\begin{array}{l}
\mathsf{conflicts}_{\mathsf{aux}} : \mathbb{N} \to \mathsf{rule\_set} \to \mathsf{list}\ (\mathbb{N} \times \mathbb{N}) \\
\mathsf{conflicts}_{\mathsf{aux}}\ n\ \mathsf{nil} := \mathsf{nil} \\
\mathsf{conflicts}_{\mathsf{aux}}\ n\ (r{::}\vec{s}) \\
\quad := (\mathsf{rrs\_conflicts}\ n\ (n+1)\ r\ \vec{s}) +\!\!+ (\mathsf{conflicts}_{\mathsf{aux}}\ (n+1)\ \vec{s}).
\end{array}$$

where $+\!\!+$ is the append operator on lists. So, in the case of a nonempty list, the result is comprised of two parts: the conflicts between the head of the list and its tail, and the conflicts within the tail.

Finally, the conflict detection program just applies the function $\mathsf{conflicts}_{\mathsf{aux}}$ to the whole rule set:

$$\mathsf{find\_conflicts}\ \vec{r} := \mathsf{conflicts}_{\mathsf{aux}}\ 0\ \vec{r}.$$

We assumed that the numbering of the rules in the rule set starts with 0. Often the convention is to start with a first rule with number 1; in that case we just need to replace 0 with 1 in the definition of find_conflicts. For the correctness proof to carry over, we would also need to modify accordingly the function Nth and replace $<$ with $\leq$ in the definition of rs_conflict.

We can optimize the algorithm further by programming a tail-recursive version of it. This will prevent a stack overflow for very large rule sets. The transformation to a tail-recursive form is a standard technique.

$$\begin{array}{l}
\mathsf{rrs\_confl}_{\mathsf{tl}} :\ \ \mathbb{N} \to \mathsf{list}\ (\mathbb{N} \times \mathbb{N}) \to \mathbb{N} \to \\
\qquad\qquad\qquad \mathsf{access\_rule} \to \mathsf{rule\_set} \to \mathsf{list}\ (\mathbb{N} \times \mathbb{N}) \\
\mathsf{rrs\_confl}_{\mathsf{tl}}\ i\ l\ n\ r\ \mathsf{nil} := l \\
\mathsf{rrs\_confl}_{\mathsf{tl}}\ i\ l\ n\ r\ (h{::}\vec{s}) \\
\quad := \ \mathbf{if}\ (\mathsf{conflict\_check}\ r\ h) \\
\qquad\quad \mathbf{then}\ (\mathsf{rrs\_confl}_{\mathsf{tl}}\ i\ ((i, n){::}l)\ (n+1)\ r\ \vec{s}) \\
\qquad\quad \mathbf{else}\ (\mathsf{rrs\_confl}_{\mathsf{tl}}\ i\ (n+1)\ r\ \vec{s}); \\
\\
\mathsf{confl}_{\mathsf{aux,tl}} : \mathsf{list}\ (\mathbb{N} \times \mathbb{N}) \to \mathbb{N} \to \mathsf{rule\_set} \to \mathsf{list}\ (\mathbb{N} \times \mathbb{N}) \\
\mathsf{confl}_{\mathsf{aux,tl}}\ l\ n\ \mathsf{nil} := l \\
\mathsf{confl}_{\mathsf{aux,tl}}\ l\ n\ (r{::}\vec{s}) \\
\quad := \mathsf{confl}_{\mathsf{aux,tl}}\ (\mathsf{rrs\_confl}_{\mathsf{tl}}\ n\ l\ (n+1)\ r\ \vec{s})\ (n+1)\ \vec{s}; \\
\\
\mathsf{find\_confl}_{\mathsf{tl}} : \mathsf{rule\_set} \to \mathsf{list}\ (\mathbb{N} \times \mathbb{N}) \\
\mathsf{find\_confl}_{\mathsf{tl}}\ \vec{r} := \mathsf{confl}_{\mathsf{aux,tl}}\ \mathsf{nil}\ 0\ \vec{r}.
\end{array}$$

It is easy to prove that the tail-recursive version of the algorithm is equivalent to the original formulation. The only difference is that the results are given in inverse order.

THEOREM 1 (find_conflicts_tl_equivalence).

$$\forall \vec{s} : \mathsf{rule\_set}, \mathsf{find\_conflicts}\ \vec{s} = \mathsf{reverse}\ (\mathsf{find\_confl}_{\mathsf{tl}}\ \vec{s}).$$

In Sections 6 and 7, we prove soundness and completeness, which, as argued, establish the correctness of our algorithm. These proofs are formalized for the algorithm find_conflicts and, by the previous theorem, they extend immediately to $\mathsf{find\_confl}_{\mathsf{tl}}$. The next section is dedicated to a description of the extracted OCaml program.

5. THE EXTRACTED PROGRAM

The interaction between programming and formal theorem proving can be realized with two distinct methodologies: the common *external* approach and the constructive *integrated* approach [12, 9]. The traditional external approach consists of implementing an algorithm as a program in some programming language and then using a logical system to prove its correctness. The integrated approach, on the other hand, exploits the computational content of constructive logic to extract a program from the proof of a theorem.

Since Coq is not just a proof-assistant, but also a functional programming language, we could program our algorithm directly in it. Then the proof of correctness guarantees that, when we run the Coq functions find_conflicts and find_confl$_{\mathrm{tl}}$, the results are provably correct. For large rule bases, however, the Coq implementation may not be efficient enough and we may need to extract the program to a more conventional programming language.

Coq has an extraction mechanism that produces an OCaml program from any Coq object [24, 18]. Since Coq developments may contain logical information that has no counterpart in OCaml, the extraction mechanism deletes all non-computational parts of the Coq objects.

A further strength of Coq is that executable programs can also be extracted from constructive proofs. These are proofs whose types live in the sort Set rather than in Prop. Indeed, our initial development did not contain a direct implementation of the boolean decision functions, but used instead constructive decidability results.

For example, instead of using the boolean function

$$\mathsf{ip\_address\_bool} : \mathsf{ip\_address} \rightarrow \mathsf{ip\_address} \rightarrow \mathbb{B},$$

we would use a lemma

$$\begin{array}{l}\mathsf{ip\_address\_dec} : \\ \forall a_1\ a_2 : \mathsf{ip\_address}, \{\mathsf{ip\_match}\ a_1\ a_2\} + \{\neg(\mathsf{ip\_match}\ a_1\ a_2)\},\end{array}$$

which states that the predicate ip_match is decidable. The proof of such a lemma is constructive and therefore a decision procedure can be obtained from it. However, the extracted program was not efficient, so we decided to implement directly an algorithm for conflict detection, rather than obtaining it from a proof. In other words, we used the integrated approach to produce a prototype, and then switched to an external approach to realize some optimizations of the software.

Now we can again use extraction to get an OCaml program. This looks identical to the Coq implementation, except for some changes in notation, since we have used no logical information in defining find_conflicts. But there is still some inefficiency caused by the fact that Coq data-types are translated together with the functions. This means that the OCaml program does not use the native types for natural numbers, integers, bytes, Cartesian products, lists, etcetera, but uses instead the translations of Coq's inductive types. To improve efficiency, we modified the OCaml program by replacing Coq types with native OCaml types.

For example, port numbers are defined as integers. The extracted program defines a type of binary integers that is obtained from the Coq inductive implementation:

```
type positive =
  | XI of positive
  | XO of positive
  | XH

type z =
  | ZO
  | Zpos of positive
  | Zneg of positive
```

Here we have a type `positive` of positive integers with three constructors: `XH` for 1, `XO` for the function $n \mapsto 2*n$, and `XI` for the function $n \mapsto 2*n+1$. Then the integers `z` are defined by taking two copies of `positive`, one for the positive and the other for the negative numbers, plus a constant `ZO` for 0. All the basic operations on integers are also extracted from the Coq versions.

Clearly this is an inefficient excess, since OCaml has a native type of integers `int` with all the basic operations efficiently implemented. Therefore, we used `int` in place of `z` in the conflict detection program. Naturally, the proof of correctness is still valid, provided that the computational behaviours of the two integer types are equivalent. The major potential source of error in this replacement is that, while elements of `z` are arbitrary precision integers, elements of `int` are 31-bit integers. This would cause an error if numbers with more that 31 bits are used. Fortunately, in our development `z` is used to model two entities: bytes, which have only 8 bits, and port numbers, which have 16 bits. Therefore no error can ever occur.

To demonstrate that the OCaml version of the program is just a reformulation of the Coq algorithm in a different syntax, here is, as example, the extracted boolean function checking whether there is a conflict between two rules:

```
let conflict_check r1 r2 =
  let { r_action = a1;
        r_protocol = pt1;
        r_source_ip = sip1;
        r_source_pn = spn1;
        r_dest_ip = dip1;
        r_dest_pn = dpn1 } = r1
  in
  let { r_action = a2;
        r_protocol = pt2;
        r_source_ip = sip2;
        r_source_pn = spn2;
        r_dest_ip = dip2;
        r_dest_pn = dpn2 } = r2
  in
  a1<>a2 && pt1=pt2 &&
  ip_range_int sip1 sip2 &&
  port_number_int spn1 spn2 &&
  ip_range_int dip1 dip2 &&
  port_number_int dpn1 dpn2
```

Compare it with the type-theoretic version of the previous section. It is basically the same, although Coq automatically performed a simplification of the (**if** – **then** – **else** –) construct before the extraction.

In the OCaml program we used, for efficiency, the tail-recursive version of the conflict detection algorithm.

To give an idea of the efficiency of the algorithm, below are the execution times (on an IBM ThinkPad X60s) for some sample rule sets. The generator for these test rule sets is included in the file `conflicts.ml`: the command `rs_generate z1 z2 z3 z4` will generate a rule set containing $5(z_1+1)(z_2+1)(z_3+1)(z_4+1)$ rules. The smallest set (when all arguments are 0) is a list of 5 rules containing 3 conflicts.

num. of rules	conflicts found	exec. time
5000	5600	4s
10000	15200	14s
20000	43200	54s
40000	86400	217s
100000	216000	23m
200000	432000	92m

6. PROOF OF SOUNDNESS

Now we want to prove the soundness of the algorithm. Intermediate results guarantee that the boolean functions used to check the intersection of the ranges of the fields of rules are sound with respect to the meaning given to them.

The boolean check function for IP ranges gives true if and only if the two ranges have a non-empty intersection, that is, if there exists an IP address that matches both:

Lemma 1 (ip_range_intersection).

$$\forall\, r_1\ r_2 : \mathsf{ip\_range},\ (\mathsf{ip\_address\_bool}\ r_1\ r_2) = \mathsf{true} \rightarrow \exists a : \mathsf{ip\_address}, (\mathsf{ip\_match}\ a\ r_1) \wedge (\mathsf{ip\_match}\ a\ r_2).$$

Proof. The proof of this lemma is based on similar results at the level of bits and bytes and it is quite straightforward. □

Similar lemmas hold for the other fields. Once we put them all together, we can prove the correctness of the conflict check function:

Lemma 2 (conflict_check_soundness).

$$\forall r_1\ r_2 : \mathsf{access\_rule},\ \mathsf{conflict\_check}\ r_1\ r_2 = \mathsf{true} \rightarrow \mathsf{rule\_conflict}\ r_1\ r_2.$$

Finally, the proof of soundness of the conflict detection algorithm requires only some bookkeeping on the indices.

Theorem 2 (conflicts_soundness).

$$\forall (rs : \mathsf{rule\_set})(i\ j : \mathbb{N}),\ (i,j) \in (\mathsf{find\_conflicts}\ rs) \rightarrow \mathsf{rs\_conflict}\ rs\ i\ j.$$

7. COMPLETENESS

Besides proving that all the pairs detected by find_conflicts are actual conflicts, we must also guarantee that all conflicts are discovered by the program. That is, we need to establish completeness. All properties of the boolean check functions that were proved in the previous section can be reversed. For example, we can prove that if two port number ranges overlap, then the corresponding check function gives true:

Lemma 3 (intersection_pn_range).

$$\forall (r_1\ r_2 : \mathsf{pn\_interval})(n : \mathsf{port\_number}),\ \mathsf{pn\_match}\ n\ r_1 \rightarrow \mathsf{pn\_match}\ n\ r_2 \rightarrow (\mathsf{pn\_bool}\ r_1\ r_2) = \mathsf{true}.$$

In this statement we changed the existential quantification on n in the hypothesis into an equivalent global universal quantification, to simplify the logic. Similar lemmas hold for the other fields. Using such results, we prove the completeness theorem for the conflict detection program.

Theorem 3 (conflicts_completeness).

$$\forall (\bar{r} : \mathsf{rule\_set})(i\ j : \mathbb{N}),\ i < j \rightarrow \mathsf{rs\_conflict}\ \bar{r}\ i\ j \rightarrow (i,j) \in (\mathsf{find\_conflicts}\ rs).$$

8. RELATED WORK

As previously mentioned, numerous other firewall analysis algorithms and tools have been developed, some of which can perform inconsistency and redundancy checks, but none of those that we consulted has been formally verified. We mention a few related approaches here. Tools that allow user queries for the purpose of analysis and management of firewall rules include Firmato [5] and Lumeta [31]. Lumeta is a successor to Fang [23] with capabilities for automatically generating queries that highlight risks.

Tools that provide analyses which detect inconsistencies are most closely related to our approach, especially those whose analyses involve a pairwise comparison of rules, such as the work of Eronen and Zitting [11] on the implementation of an expert system and of Al-Shaer and Hamed [2, 3] on the Firewall Policy Advisor tool. In all the literature that we examined, either it is not clear how masks are handled, or it is assumed that a mask specifies a single interval. Although this is often the case in practice, our example showed that a single mask can specify a range comprising multiple intervals of IP addresses. We incorporated full general ranges into our definitions. In addition, because of the success of our performance tests, we believe that our approach is an optimization on the specific class of problems that can be handled by all these approaches. Although it is not possible to compare directly, since we don't have a copy of their implementations, statistics given for the Firewall Policy Advisor consider up to 100 rules. Statistics are not given for Eronen and Zitting's expert system, but we did a cursory comparison by also implementing and testing a prototype of both their approach and our approach in Prolog. The Firewall Policy Advisor tool also includes a high-level policy editor which helps a firewall administrator avoid introducing new conflicts when adding rules.

Other algorithms that involve pairwise comparisons are those of Hari et. al. [16], Eppstein and Muthukrishnan [10], and Baboescu and Varghese [4]. These algorithms assume a rule format with no masks and a simplified kind of range, which allows for efficient algorithms which scale well. These papers report good experimental results; in some cases up to tens of thousands of rules are processed successfully. The FIREMAN toolkit [32] is another example of this approach to detecting inconsistencies and redundancies in single firewalls and in networks of firewalls. The set of all possible requests is formulated and model checking is used to divide the set into those which are accepted, those which are rejected, and those for which no rule applies. The experimental results include firewalls of up to 800 rules, but the tool could likely handle more since it uses an efficient BDD representation.

The ACLA framework [25] is another set of algorithms for performing various analyses on firewall specifications. These analyses include detecting conflicts, as well as others such as redundancy checking. The main aim of the approach is to use the results of analysis to optimize firewalls. For example, if there is a conflict because two rules containing intersecting port number ranges, then these two rules are split into three rules, one for the non-intersecting part of the permit rule, one for the non-intersecting part of the deny rule, and one for the intersection of the two ranges. The order of the two rules is important for determining whether the latter rule is designated a permit rule or a deny rule; it is given the same designation as the one appearing first in the list. The result of the set of analyses is a new list of rules with the same behavior as the original with all conflicts and redundancies removed. While this kind of analysis can detect conflicts, the goal is different from ours. The ACLA approach assumes that the firewall has no errors and returns an optimized version that could be quite different from the original, and thus hard to read for the original programmer. Our approach reports conflicts to a firewall administrator for help in debugging a firewall specification.

Another algorithm for both reporting and removing conflicts and redundancies is presented in [8]. The kinds of transformations done are similar to those of the ACLA framework, though the algorithms differ. In [8], redundancies and conflicts are both removed automatically and reported to the system administrator.

Liu and Gouda [19] give an algorithm specialized for finding redundancies. Like in the ACLA work, they assume that the input firewalls are correct. Under such an assumption, a redundancy is any rule such that the operation of the firewall would be the same without it, which differs significantly from our definition (and that of [32]). For example, given two rules that cover the same set of packets, the one occurring later in the list of rules is considered redundant, even if one rule accepts these packets, and the other denies them; this is a conflict in our case. On the other hand, their definition does not cover rules which both deny or both accept a set of packets in the case when the set of packets covered by one rule is not a subset of the packets covered by the other; our definition does cover this case. Thus the algorithms developed are necessarily different from ours. No experimental results are given for this technique.

Guttman and Herzog [13, 14] develop a high-level model of a network and a set of algorithms and tools for analyzing both firewall rules and network intrusion detection systems. This work focuses on ensuring that firewall and network configurations satisfy a given policy; the analyses they can perform include detecting inconsistencies and redundancies but are more general. Uribe and Cheung [30] also use similar modeling techniques and perform a variety of analyses. The Network Policy Enforcement tool [14] is one of the more recent tools in this line of work. It uses a BDD representation and reports successful results on 1,300 access rules in a single run, and is likely to work for even larger rule sets.

9. CONCLUSION

The results presented here illustrate that it is possible to write a conflict detection algorithm for firewalls that is both formally verified and efficient. When we began this work, we first developed a method to prove, for each firewall specification, that it was conflict-free. The main advantage of the method shown here, i.e., proving the correctness of the program which detects conflicts, is clear. There is a single proof required; we proved the correctness once and for all, and then can apply the verified extracted program to as many firewalls as needed. We have also reported that our algorithm is efficient. In particular, we successfully applied our conflict detection implementation to a firewall with over 200,000 rules.

In papers where classification systems are given, conflicts are divided into several categories. For example, in [32], *shadowing* refers to a conflict where the packets covered by a rule appearing later in the list is a subset of the packets covered by an earlier rule, *generalization* is the reverse where the rule with the subset appears first, and *correlation* covers the case where the packets covered by both rules intersect, but neither is a subset of the other. Our algorithm covers all these cases, but does not distinguish them. We could easily add such a distinction to the algorithm and to the reporting of conflicts. Also, as noted, although we have not done so, our techniques could easily be applied to detect redundancies as well, and we could also specify the corresponding three categories of redundancies.

The kind of reasoning needed to complete the formal proof was not deep, and more of the proof could likely be automated if we were to use a system that implements a less expressive logic and has more automated reasoning capabilities. We chose Coq for a variety of reasons. For example, its expressive power allowed a direct and elegant formulation of definitions, leading to a concise proof. Also, it has a program extraction facility that is direct and easy to use. The fact that the proof was done interactively was not a big cost since the proof was done once. Coq does provide facilities to develop automated procedures which we could apply to this particular application. If we were to consider a wider class of firewall analysis tools, each which needed to be proved correct independently, it would likely become worthwhile to further investigate this potential as well as compare it with the use of other systems which provide more automation directly.

Our future work will include handling blanket rules. As mentioned, a blanket rule expresses constraints on many hosts or port numbers at once, specifying a default policy that may be contradicted by more specific rules appearing earlier. For example, a firewall administrator may want to deny FTP for all ports numbers from 20 to 30 but permit FTP for a given destination host from any source. The permit rule would appear before the blanket deny rule. Thus, even though these rules describe opposite behavior for one destination host, they should not in principle be considered conflicting because they implement the intended behavior. Handling such rules will involve formally defining the notion of blanket, and incorporating it into our definition of conflict.

It would be interesting to try to formally verify the analysis algorithms of FIREMAN, ACLA, and several other systems, which could both confirm their correctness as well as provide a method of formal comparison of the definitions of conflict used by the different approaches. In addition, other areas such as firewall policy deployment are important in practice and can involve complex algorithms (e.g. [33]) which may benefit from formal verification.

Our work paves the way for considering more general policy representation languages like XACML [28], which is becoming increasingly important in the security domain. Some work on detecting conflicts in XACML policies can be found in [20, 21]. We expect to be able to apply our method for handling ranges such as those specified by host/mask pairs to a variety of other kinds of data which are used by XACML to express both numerical and non-numerical constraints. We have so far implemented a prototype in Prolog for a large subset of XACML, which will serve as the starting point for a formal version.

Acknowledgements

The authors are grateful to Luigi Logrippo and to Jacques Sincennes for their input to this research. The work described here is supported by the Ontario Centres of Excellence and the Natural Sciences and Engineering Research Council of Canada.

10. REFERENCES

[1] Ehab Al-Shaer, Hazem Hamed, Raouf Boutaba, and Masum Hasan. Conflict classification and analysis of distributed firewall policies. *IEEE Journal on Selected Areas in Communications*, 23(10):2069–2084, October 2005.

[2] Ehab S. Al-Shaer and Hazem H. Hamed. Firewall policy advisor for anomaly discovery and rule editing. In *IFIP/IEEE Eighth International Symposium on Integrated Network Management*, pages 17–30, 2003.

[3] Ehab S. Al-Shaer and Hazem H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proceedings of IEEE Infocom*, volume 4, pages 2605–2626, 2004.

[4] Florin Baboescu and George Varghese. Fast and scalable conflict detection for packet classifiers. In *Tenth IEEE International Conference on Network Protocols*, pages 270–279, 2002.

[5] Yair Bartal, Alain Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. *ACM Transactions on Computer Systems*, 22(4):381–420, November 2004.

[6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[7] James Boney. *CISCO IOS in a Nutshell*. O'Reilly, first edition, 2001.

[8] F. Cuppens, N. Cuppens-Boulahia, and J. García-Alfaro. Detection and removal of firewall misconfiguration. In *IASTED International Conference on Communication, Network, and Information Security*, 2005.

[9] P. Dybjer. Comparing integrated and external logics of functional programs. *Science of Computer Programming*, 14:59–79, 1990.

[10] David Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 827–835, 2001.

[11] Pasi Eronen and Jukka Zitting. An expert system for analyzing firewall rules. In *6th Nordic Workshop on Secure IT Systems*, pages 100–107, 2001.

[12] J.-Y. Girard. Linear logic and parallelism. In *Mathematical Models for the Semantics of Parallelism*, volume 280 of *Lecture Notes in Computer Science*, pages 166–182. Springer-Verlag, 1986.

[13] Joshua D. Guttman. Filtering postures: Local enforcement for global policies. In *IEEE Symposium on Security and Privacy*, pages 120–129, 1997.

[14] Joshua D. Guttman and Amy L. Herzog. Rigorous automated network security management. *International Journal of Information Security*, 2003. To appear.

[15] Hazem Haded and Ehab Al-Shaer. Taxonomy of conflicts in network security policies. *IEEE Communications Magazine*, 44(3):134–141, March 2006.

[16] Adiseshu Hari, Subhash Suri, and Guru Parulkar. Detecting and resolving packet filter conflicts. In *Proceedings of IEEE Infocom*, volume 3, pages 1203–1212, 2000.

[17] W. A. Howard. The formulae-as-types notion of construction. In J. P. Selding and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[18] Pierre Letouzey. A new extraction for Coq. In *Types for Proofs and Programs, Second International Workshop*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer-Verlag, 2003.

[19] Alex X. Liu and Mohamed G. Gouda. Complete redundancy detection in firewalls. In *Data and Applications Security*, volume 3654 of *Lecture Notes in Computer Science*, pages 196–209. Springer-Verlag, 2005.

[20] Mahdi Mankai. Vérification et analyse des politiques de contrôle d'accès: Application au langage XACML. Master's thesis, Université du Québec en Outaouais, January 2005.

[21] Mahdi Mankai and Luigi Logrippo. Access control policies: Modeling and validation. In K. Adi, D. Amyot, and L. Logrippo, editors, *5th NOTERE Conference (Nouvelles Technologies de la Répartition)*, pages 85–91, Gatineau, Canada, 2005.

[22] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

[23] Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A firewall analysis engine. In *IEEE Symposium on Security and Privacy*, pages 177–187, 2000.

[24] Christine Paulin-Mohring. Extracting F(omega)'s programs from proofs in the calculus of constructions. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 89–104, 1989.

[25] Jiang Qian, Susan Hinrichs, and Klara Nahrstedt. ACLA: A framework for access control list (ACL) analysis and optimization. In *Proceedings of Communications and Multimedia Security*, 2001.

[26] Jeff Sedayao. *Cisco IOS Access Lists*. O'Reilly, June 2001.

[27] Morten Heine B. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier Science, 2006.

[28] Sun Microsystems. A brief introduction to XACML. http://www.oasis-open.org/committees/-download.php/2713/-Brief_Introduction_to_XACML.html, 2003.

[29] The Coq Development Team. LogiCal Project. *The Coq Proof Assistant. Reference Manual. Version 8*. INRIA, 2004. Available at the web page `http://pauillac.inria.fr/coq/coq-eng.html`.

[30] Tomás E. Uribe and Steven Cheung. Automatic analysis of firewall and network intrusion detection system configurations. In *ACM Workshop on Formal Methods in Security Engineering*, pages 66–74, 2004.

[31] Avishai Wool. Architecting the Lumeta firewall analyzer. In *10th USENIX Security Symposium*, 2001.

[32] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In *IEEE Symposium on Security and Privacy*, 2006.

[33] Charles C. Zhang, Marianne Winslett, and Carl A. Gunter. On the safety and efficiency of firewall policy deployment. In *IEEE Symposium on Security and Privacy*, pages 33–50, 2007.