

Available online at www.sciencedirect.com**SciVerse ScienceDirect**

ELSEVIER

journal homepage: www.elsevier.com/locate/cose
**Computers
&
Security**


CrossMark

Management of stateful firewall misconfiguration

Joaquin Garcia-Alfaro ^{a,*}, Frédéric Cuppens ^b, Nora Cuppens-Boulahia ^b,
 Salvador Martinez ^c, Jordi Cabot ^c

^a Institut Mines-Telecom, Telecom SudParis, CNRS Samovar UMR 5157, 9 rue Charles Fourier, 91011 Evry, France

^b Institut Mines-Telecom, Telecom Bretagne, 35576 Cesson-Sévigné, France

^c Atlanmod, Ecole des Mines de Nantes, Inria, Lina, Nantes, France

ARTICLE INFO

Article history:

Received 28 November 2012

Received in revised form

9 January 2013

Accepted 12 January 2013

Keywords:

Network security

Access control

Firewalls

Stateless rules

Stateful rules

Iptables

Netfilter

Misconfiguration

Anomalies

Model-driven engineering

ABSTRACT

Firewall configurations are evolving into dynamic policies that depend on protocol states. As a result, stateful configurations tend to be much more error prone. Some errors occur on configurations that only contain stateful rules. Others may affect those holding both stateful and stateless rules. Such situations lead to configurations in which actions on certain packets are conducted by the firewall, while other related actions are not. We address automatic solutions to handle these problems. Permitted states and transitions of connection-oriented protocols (in essence, on any layer) are encoded as automata. Flawed rules are identified and potential modifications are provided in order to get consistent configurations. We validate the feasibility of our proposal based on a proof of concept prototype that automatically parses existing firewall configuration files and handles the discovery of flawed rules according to our approach.

© 2013 Elsevier Ltd. All rights reserved.

1. Introduction

Firewalls aim at optimising the degree of security deployed over an information system. Their configuration is, however, very complex and error-prone. It is based on the distribution of several packages of security rules that define properties such as acceptance and rejection of traffic. The assembly of all these properties must be consistent, addressing always the same decisions under equivalent conditions, and avoiding conflicts or redundancies. Otherwise, the existence of anomalies and misconfiguration will lead to weak security architectures, potentially easy to be evaded by unauthorised

parties. Approaches based on formal refinement techniques, e.g., using abstract machines grounded on the use of set theory and first order logic, ensures, by construction, cohesion, completeness and optimal deployment (Preda et al., 2010). Unfortunately, these approaches have not always a wide follow. Network policies are often empirically deployed over firewalls based on security administrator expertise and flair. It is then relevant to analyse these deployed configurations in order to detect and correct errors, known in the literature as misconfiguration discovery. Several research works exist to directly manage the discovery and correction of stateless firewall configuration anomalies (Garcia-Alfaro et al., 2008; Hari

* Corresponding author. Tel.: +33 160 76 47 55.

E-mail address: joaquin.garcia-alfaro@acm.org (J. Garcia-Alfaro).
 0167-4048/\$ – see front matter © 2013 Elsevier Ltd. All rights reserved.
<http://dx.doi.org/10.1016/j.cose.2013.01.004>

et al., 2000; Al-Shaer and Hamed, 2004; Yuan et al., 2006). By stateless firewall configurations we refer to the security policies of first generation firewalls, mostly packet filtering devices working only on the lower layers of the OSI reference model. In this paper, we are particularly interested in addressing the analysis of deployed configurations of second and third generation firewalls peeking into the transport and upper layers.

The main goal of a firewall is to control network traffic flowing across different areas of a given local network. It must provide either hardware or software means to block unwanted traffic, or to re-route packets towards other components for further analysis. In the stateless case, the filtering actions, such as accepting or rejecting packet flows, are taken according to a set of static configuration rules. These rules only pay attention to information contained in the packet itself, such as network addresses (source and destination), ports and protocol. The main advantage of stateless firewalls is their filtering operations speed. However, since they do not keep track of state connection data, they fail at handling some vulnerabilities that benefit from the position of a packet within existing streams of traffic. Stateful firewalls solve this problem and improve packet filtering by keeping track of connection status. Indeed, they can block those packets that are not meeting the valid state machine of a given connection-oriented protocol. As with stateless packet filtering, stateful filtering intercepts the packets at the network layer and verifies if they match previously defined security rules. Moreover, stateful firewalls keep track of each connection in an internal state table. Although the entries in this table varies according to the manufacturer of every product, they typically include source and destination IP addresses, port numbers and information about the connection status.

Most methods that have been proposed to detect anomalies in the configuration of firewalls, such as (Garcia-Alfaro et al., 2008; Hari et al., 2000; Al-Shaer and Hamed, 2004; Yuan et al., 2006), are limited to the stateless case. Little work has been done for the detection of anomalies in the stateful case. Some approaches aim at describing stateful firewall models (Gouda and Liu, 2005), while others adapt management processes previously designed for stateless firewalls (Buttyan et al., 2009). In Cuppens et al. (2012), we uncovered new misconfiguration types that lead to flawed configurations in which some stateful actions, according to a connection-oriented protocol, are conducted by the firewall, while other related actions are not. We also provided algorithmic solutions to discover and correct explicit conflicting rules, so that the resulting set gets consistent with the action of those rules with higher priority in order. In this paper,¹ we complement the algorithmic solutions in Cuppens et al. (2012), in order to assist and guide in the correction of the more complex case, in which misconfiguration is driven by the omission of explicit rules in a policy. The principle of our approach is based on the specification of general automata. Such automata describe the different states that traffic packages can take throughout the filtering process. We also present the ongoing development of

a proof of concept prototype that shows the validity of our approach in the case of stateful deployed configurations. The prototype, based on model-driven engineering, extends the results presented in Martinez et al. (2012) for managing stateless configurations based on Linux firewalls, and tackles the stateful case. The model-driven engineering approach is chosen with the aim of getting rid off the low level details of the concrete solution, and provide a solution for any other stateful filtering system with minimum effort. The prototype we present also provides an extension of MIRAGE (Garcia-Alfaro et al., 2011), a firewall audit tool for the automatic detection and correction of stateless firewall configuration anomalies. The extension aims at covering the management of stateful firewalls as well.

1.1. Paper organisation

Section 2 presents in more detail our motivation scenario. Section 3 presents the case of detecting anomalies on those configurations that contain only stateful rules. Section 4 addresses the case in which stateful and stateless rules coexist in a given configuration rule set. Section 5 presents our automatic audit tool that parses deployed stateful firewall configuration files and handles the discovery of flawed rules according to our approach. Section 6 surveys related work. Section 7 closes the paper.

2. Motivation scenario

Stateful firewalls provide fine-grained filtering capabilities to protect networks against complex attacks. For instance, stateful filtering can be used to detect and block anomalous behaviour in traffic flows that progress via invalid connection states of a given connection-oriented protocol. Assume the case of the TCP protocol and its simplified automaton depicted in Fig. 1(a). The automaton describes the progression of a TCP connection exhibiting normal behaviour (Treurniet, 2006). Illicit scanning activities (Meng et al., 2008) or brute force termination attacks (Arlitt and Williamson, 2005) can be described in the configuration of a stateful firewall, so that albeit of being dropped, such activities are also reported to the security administrator. As represented by the automaton, a TCP connection progresses from state to state based on the information contained in the headers of the TCP packets exchanged between two peers, and specified as the TCP traffic flag combination in Fig. 1(b). Based on this approach, the security administrator can now signal invalid transitions, as represented in Fig. 1(c) by the symbol \emptyset . This way, illicit scanning activities and brute force termination attacks can easily be identified by means of the invalid transitions. Existing tools such as NMAP (Mapper) and HPING3 (Sool) are available on-line to conduct and verify such kind of illicit activities. Using the information in Fig. 1, the security administrator can now define a list of stateful filtering rules to report and block these invalid transitions. Exactly how this shall be done differs from one firewall to another, but let us exemplify here the case for stateful firewalls based on the Linux firewall architecture (The NetFilter Project: Fi). Such an architecture, popularly known as Netfilter, provides stateful filtering

¹ This is an expanded and revised version of Cuppens et al. (2012), published at the 27th IFIP TC-11 Information Security Conference.

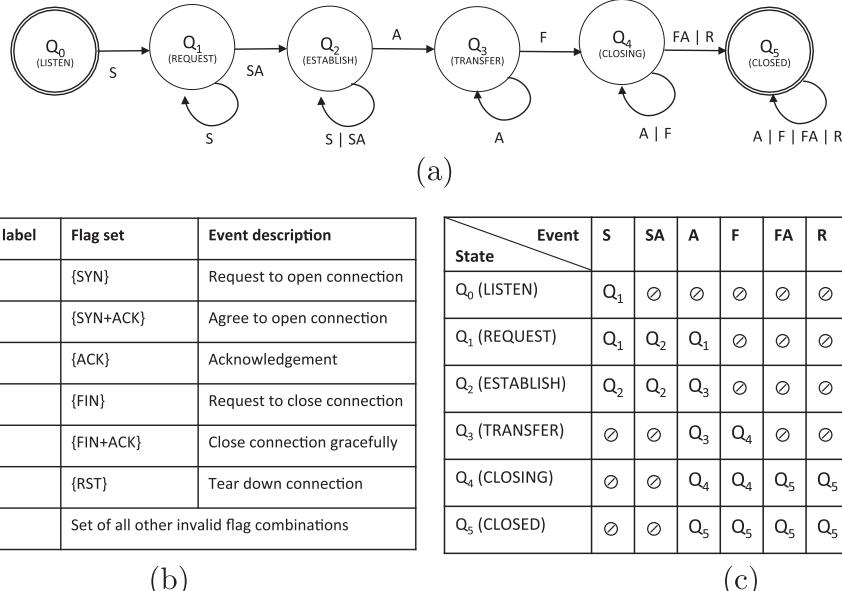


Fig. 1 – Progression of a TCP connection exhibiting normal behaviour, based on (Treurniet, 2006). (a) Unified TCP automaton (for simplicity, it represents together the two separate automata, one for the client and one for the server, of the traditional TCP finite state machine). (b) Events description. (c) Transition table, where the symbol \emptyset represents the invalid state.

capabilities in order to grant access to network traffic based on already existing connections. This feature of Netfilter is based on the use of the `iptables` and `conntrack` modules. The `iptables` module defines tables (e.g., filtering tables) and chains (e.g., input, forward and output chains) to conduct actions over packets traversing the network stack. The `conntrack` module (short for connection-tracking) provides Netfilter with the ability for maintaining state information about the packets the firewall is examining (Neira Ayuso, 2006). Therefore, the combination of both modules allows the user to define stateful filtering rules for connection-oriented protocols. Suppose the addition of rules in Listing 1 in the tables of a Linux firewall controlling connections directed towards a network server whose IP address is 5.6.7.8.

In Listing 1, rule in Line 01 sets the default policy to accept (i.e., an open policy). The rule in Line 02 creates a new chain, that will be used later by other `iptables` rules to report and drop all packets assigned to that chain (cf. Lines 04 and 05). Finally, the rule in Line 03 contains the main action. This rule is based on the `conntrack` match for `iptables`, which makes it possible to define filtering rules in a much more granular way than simply using stateless rules or rules based on the old state match. This is defined by providing the parameter `--m conntrack` to the rules

(cf. reference (The NetFilter Project: Fi) for a more extensive description of `conntrack` and (Venkatamohan, 2011) for extending the regular matching module to increase its stateful expressivity). The parameter `--ctstate NEW` is used to instruct the firewall to match those TCP packets in the `conntrack` table that are seen for the first time. The parameter `--tcp-flags`, preceded by the ‘!’ symbol, is used to exclude from such packets, those with the SYN flag header activated. Finally, the `--ctdir REPLY` parameter is used to exclude those packets originated at the IP address 5.6.7.8. As a result, the above rules allow to report and drop those TCP traffic connections across the FORWARD chain that exhibit the invalid behaviour defined in the first row of Fig. 1(c), i.e., transitions from the initial state to the invalid state, which can be associated with illicit scanning activities. In order to address the invalid transitions of the second and third rows in Fig. 1(c), we can complement now the set of rules in Listing 1 with those in Listing 2.

Equivalent rationale can be used to complement the set of rules and cover all the remainder cases of invalid transitions in Fig. 1(c). The result shall be a very granular set of stateful rules covering each of the invalid transitions. This gain in expressivity may lead, however, to error-prone configurations. Indeed, it is possible to end up with flawed rule sets, in

```

01: iptables -P FORWARD ACCEPT
02: iptables -N InvRQ
03: iptables -A FORWARD -p tcp -d 5.6.7.8 --m conntrack --ctstate NEW ! --tcp-flags ALL SYN
      --ctdir REPLY -j InvRQ
04: iptables -A InvRQ -j LOG --log-prefix 'InvRQ'
05: iptables -A InvRQ -j DROP
  
```

Listing 1. Sample Linux firewall configuration.

```

06: iptables -N InvRP
07: iptables -A FORWARD -p tcp -s 5.6.7.8 --m conntrack --ctstate ESTABLISHED --ctdir ORIGINAL
    --ctstatus SEEN_REPLY -j InvRP
08: iptables -A FORWARD -p tcp -d 5.6.7.8 --m conntrack --ctstate ESTABLISHED --ctdir REPLY
    --ctstatus ASSURED -j InvRP
09: iptables -A InvRP --tcp-flags ALL SYN -j RETURN
10: iptables -A InvRP --tcp-flags ALL SYN,ACK -j RETURN
11: iptables -A InvRP --tcp-flags ALL ACK -j RETURN
12: iptables -A InvRP -j LOG --log-prefix 'InvRP'
13: iptables -A InvRP -j DROP

```

Listing 2. Remainder rules to complete the sample Linux firewall configuration shown in Listing 1.

which some threats are not appropriately covered, while legitimate actions are denied. This can be the case when some parameters like `ctstatus`, `ctdir`, `tcp-flags`, etc., are not appropriately used. For instance, the omission (by mistake) of the ‘!’ symbol in Line 03 of our previous example would lead to a situation in which all the invalid transitions of the first row in Fig. 1(c) (e.g., illicit scanning activities) are now allowed by the firewall. At the same time, it also instructs the firewall to deny the initial connection establishment to the server, and therefore blocking all potential communication with it. Similarly, the inversion (by mistake) of values `ORIGINAL` and `REPLY` in Lines 07 and 08 in the above example would instruct the firewall to allow invalid transitions while denying valid ones. In the sequel, we address these misconfiguration issues and provide an automatic solution to handle them for any generic purpose stateful firewall system and connection-oriented protocol.

3. Intra-state rule misconfiguration

3.1. Previous work

In our previous work presented in [Cuppens et al. \(2012\)](#), we showed that a fully stateful firewall configuration (i.e., a firewall configuration containing only stateful rules) may be affected by three main misconfiguration types. The first type consists of *shadowing*, in which some rules expected to conduct a given action over the traffic are cancelled by preceding rules with higher priority in the order. As a result, some packets that should be blocked by the firewall can be granted access to reach their destination by mistake. The second type consists of *redundancy*, in which some useless rules in the configuration can be removed without changing the filtering policy of the firewall. These two first types of misconfiguration already existed for the stateless case, and can be efficiently handled by existing solutions (cf. algorithmic solutions in [Garcia-Alfaro et al. \(2006, 2008\)](#) and [Buttyan et al. \(2009\)](#)) and citations thereof, for the discovery and correction of these two cases). The third misconfiguration class, denoted as intra-state rule misconfiguration, is specific to the stateful case. The rationale assumed in our previous work reads as follows. Suppose a connection-oriented protocol, not necessarily the TCP protocol, in which we may identify (1) the establishment phase; (2) the data transfer phase; and (3) the termination phase. In such

a case, we defined that an intra-state misconfiguration arises if: (a) the client succeeds to start the handshake connection establishment with a server, while the firewall is configured in a way that some necessary steps of the handshake are rejected; or (b) the client starts the connection termination, but the firewall rejects, at least, one of the remainder or previous operations.

An algorithmic solution to discover and handle the third misconfiguration type was presented in [Cuppens et al. \(2012\)](#). The proposed solution uses automata theory in order to encode the permitted states and transitions of the protocol. Then, stateful rules are checked against the resulting automaton, in order to determine whether the initial (establishment) phase of the protocol is permitted but denied later during the remainder (e.g., transfer or termination) phases. Conflicting rules are discovered and modified, so that the resulting set gets consistent with the action with higher priority (e.g., accepting the termination phase if the establishment was accepted as well). For instance, let us assume the rule set shown in [Table 1](#). Each rule specifies an Action (e.g., `ACCEPT` or `DENY`) that applies to a set of condition attributes, such as `SrcAddr`, `DstAddr`, `SPort`, `DPort`, `Protocol`, and `Transition`. The `Transition` attribute stands for `current_state + event`, according to the automaton of the connection-oriented protocol specified in the `Protocol` attribute. Assume the state automaton depicted in [Fig. 2](#). Rule r_2 gives an example of a correction on a subset of stateful rules by using the algorithmic solution in [Cuppens et al. \(2012\)](#). In accordance with the automaton, the rule actually contradicts the decision in rule r_1 , which is allowing the initial step in the establishment of a connection. Then, the algorithm will suggest correcting rule r_2 to preserve the inner logic of the connection establishment.

Table 1 – Stateful filtering rule set addressing valid transitions. Algorithms in Cuppens et al. (2012) detect an anomaly in rule r_2 , and propose its modification from DENY to ACCEPT.

Rule	SrcAddr	DstAddr	SPort	DPort	Protocol	Transition	Action
r_1	1.2.3.4	5.6.7.8	1080	80	P	$Q_0 + E_2$	ACCEPT
r_2	5.6.7.8	1.2.3.4	80	1080	P	$Q_2 + E_3$	ACCEPT
r_3	5.6.7.8	1.2.3.4	80	1080	P	$Q_5 + E_7$	ACCEPT
r_4	1.2.3.4	5.6.7.8	1080	80	P	$Q_9 + E_8$	ACCEPT

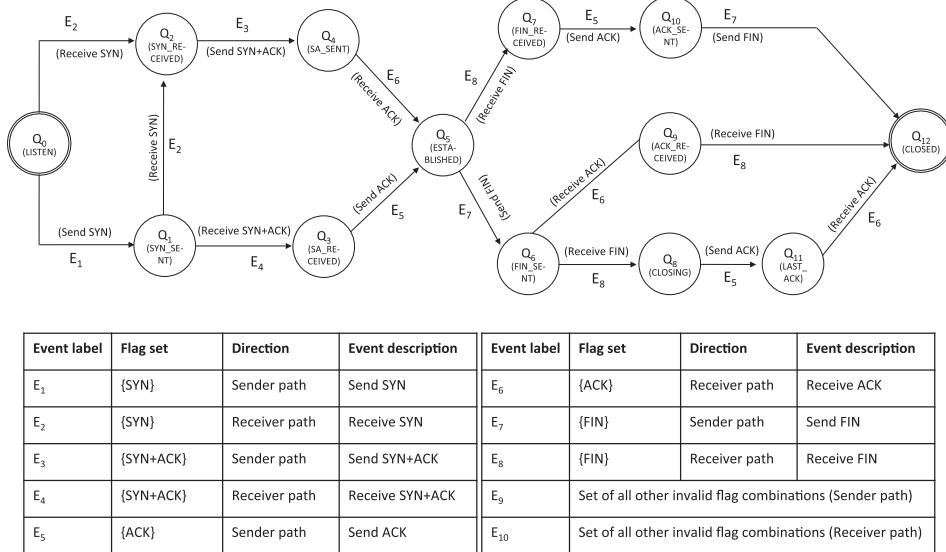


Fig. 2 – Automaton of a given connection-oriented protocol P. For simplicity, invalid transitions are not shown. Such transitions are available in Appendix A, Fig. 7.

3.2. Current limitations

A first limitation of our previous approach is that it does not warn about missing rules to fulfil complete paths of protocol automata. For instance, if we trace the path followed by rules r_1, r_2, r_3 , and r_4 , we can notice that some transitions which are necessary to allow the progression of a connection from Q_0 to Q_{12} , are not included in the rule set. A possible solution would be to warn that, at least, two extra rules covering the transitions $Q_4 + E_6$ and $Q_6 + E_6$ shall be added in the aforementioned rule set. We, therefore, need to complement the algorithmic solution provided in Cuppens et al. (2012), to verify complete coverage of automata paths by rules of a given stateful rule set. A second limitation of our previous approach is that it does not warn either about lack of coverage of rules addressing the invalid states. For instance, let us assume the rule set depicted in Table 2. This set enforces an open policy, in which we define as prohibitions those invalid transitions of the protocol that are considered either unnecessary or harmful for the network that is being protected by the stateful firewall. If we apply our previous series of algorithms in Cuppens et al. (2012), the rule set will be reported as correct. However,

notice that just a subset of those transitions to the invalid state are contained in the policy. From the list of 130 possible transitions of the automaton in Fig. 2 (i.e., 13 states times 10 events, as shown in Appendix A, Fig. 7), from which only 16 transitions are represented as valid transitions, the rule set is only covering 6 transitions (i.e., $Q_0 + E_4$ to $Q_5 + E_5$ in the Transition column of Table 2) of the remainder 114 invalid cases that shall be denied as well (see in Appendix A, Fig. 8(b) such missing rules). This case of potential misconfiguration must be treated by our approach as well. In the sequel, we extend our early algorithmic construction in Cuppens et al. (2012), and provide a more complete solution that handles these aforementioned limitations.

3.3. Extended work

Algorithm audit_rule_set complements our previous intra-state rule misconfiguration management process presented in Cuppens et al. (2012). Its pseudocode is summarised in Algorithm 1. It uses as input a stateful rule set R, in which each rule specifies an Action (e.g., ACCEPT or DENY) that applies to a set of condition attributes, such as SrcAddr, DstAddr, SPort, DPort, Transition, and Protocol. The Protocol attribute corresponds to a connection-oriented protocol. An automaton A characterising the progression of a connection for such a protocol is also provided to the algorithm. Finally, the identifiers for the initial (Q_0), final (Q_n) and invalid (\emptyset) states are also used as input parameters. The main steps of the algorithm are:

- Build a set S containing all possible paths of valid transitions connecting the initial (Q_0) and the final (Q_n) states of automaton A (Line 6);
- Build a set T containing all the transitions of automaton A to reach the invalid (\emptyset) state (Line 7);

Table 2 – Stateful filtering rule set addressing invalid transitions. Algorithms in Cuppens et al. (2012) report the set as correct.

	Rule	SrcAddr	DstAddr	SPort	DPort	Protocol	Transition	Action
r_1		1.2.3.4	5.6.7.8	1080	80	P	$Q_0 + E_4$	DENY
r_2		5.6.7.8	1.2.3.4	80	1080	P	$Q_1 + E_1$	DENY
r_3		1.2.3.4	5.6.7.8	1080	80	P	$Q_2 + E_2$	DENY
r_4		5.6.7.8	1.2.3.4	80	1080	P	$Q_3 + E_3$	DENY
r_5		1.2.3.4	5.6.7.8	1080	80	P	$Q_4 + E_8$	DENY
r_6		5.6.7.8	1.2.3.4	80	1080	P	$Q_5 + E_5$	DENY

- Verify the coverage of either S or T (Lines 8–20), w.r.t. the Action (i.e., ACCEPT or DENY) and Transition attributes of all the rules in R;
 - In case of a permission (i.e., rule in R whose Action attribute is set to ACCEPT) covering one of the transitions in a given path of S, verify that all the remainder transitions of such a path are also covered by other permissions in R (Lines 11 and 15). If the verification fails, report those transitions that are not covered and the necessary rules to correct the failure (Lines 16 and 18).
 - In case of a prohibition (i.e., rule in R whose Action attribute is set to DENY) covering, at least, one of the transitions in T, verify that all the other transitions in T are also covered in R (Lines 13 and 15). If the verification fails, report those transitions that are not covered and the necessary rules to correct the failure (Lines 16 and 18).

```

1: Input: A, protocol automaton
2: Input: R, stateful rule set
3: Input: Q0, initial state of automaton A
4: Input: Qn, final state of automaton A
5: Input: ∅, invalid state of automaton A
6: S ← find_all_paths(A,Q0,Qn);
   /* S: set of sets, s.t. every element in S is a set of valid transitions connecting Q0 and Qn */
7: T ← transitions_to_state(A,∅);
   /* T: set of invalid transitions, s.t. every element in T is a transition of A to reach ∅ */
8: repeat
9:   ri ← next_unvisited_rule(R);
10:  if (ri[Action] = ACCEPT) then
11:    L ← prune_paths(S,ri[Transition]);
        /* L ⊆ S, s.t. every path in L is a set of valid transitions connecting Q0 and Qn that
           contains the state represented by ri[Transition] */
12:  else if (ri[Action] = DENY) then
13:    L ← T;
14:  end if
15:  C ← cover_with_rules(L,R,ri);
16:  M ← extract_missing_rules(C);
        /* M: set of mutually disjoint rules derived from R, s.t. M ∩ R = ∅; let m be a rule in M,
           then m[Action] = ri[Action], Address and Port attributes of m are consistent with rules in R, and
           m[Transition] is necessary to fully cover the set of transitions in L */
17:  if (M ≠ ∅) then
18:    report('Intra-state misconfiguration in R discovered via ri. Update R based on rules in M');
19:  end if
20: until no_more_rules_to_visit(R);
   /* no-more-rules-to-visit(R) holds true whenever all rules in R but the last are reported as visited (the
      last rule is not inspected given that it is just a mark to the default policy) */

```

Algorithm 1. audit_rule_set (**A, R, Q₀, Q_n, ∅**).

Let us elaborate further on the use of **Algorithm 1** by describing the example shown in **Fig. 3**. For simplicity, we assume here just the case of a closed policy (rule set with only permissions, whose default policy is set to block all those packets not matching any given permission). An example to handle as well open policies (rule set with only prohibitions, whose default policy is set to grant access to all those packets not matching any given permission) is provided in **Appendix A**. In both examples, we assume that the initial rule sets have previously been processed by the series of algorithms in **Garcia-Alfaro et al. (2008, 2006)**. This way, we can guarantee that the rule sets are free of shadowing and redundancy, and

that the rules are mutually disjoint. These algorithms can be applied in the stateful case as well (as pointed out in **Butyan et al. (2009)**) and allow us to transform rule sets with mixed policies (combining both permissions and prohibitions) in a closed or open way as well. Notice, however, that the application of our extended solution does not alter the configurations. It correlates rules with regard to network layer information and reports those missing data with regard to stateful coverage, to detect the existence of intra-state rule misconfiguration.

The first step of the algorithm is the construction of sets S and T. Assume an automaton A based on the finite state machine depicted in **Fig. 2**. For simplicity, the graphical representation does not contain the invalid state. The complete table of transitions, containing the invalid state, is shown in **Appendix A, Fig. 7**. To build S, the algorithm uses function **find_all_paths**. Function **find_all_paths** recursively performs exhaustive search of automaton A and keeps track of all the possible paths

of valid transitions to go from Q₀ to Q₁₂. More precisely, it starts at node Q₀ and builds a new subset for S each time it reaches Q₁₂. **Fig. 3(b)** shows a sample interpretation of S over a two-dimensional vector. Similarly, T is built using function **transitions_to_state**. This function starts at ∅ (invalid state) and recursively builds T with all the transitions it finds over a one-dimensional vector (cf. **Appendix A, Fig. 9**). Assume now the rule set shown in **Fig. 3(a)**. This rule set contains a closed policy, i.e., it contains only permissions. The processing of this rule set starts in Line 9, based on function **next_unvisited_rule**. This function processes the rules in R as a list, and returns those unmarked ones, one at each execution time. All the rules are

Rule	SrcAddr	DstAddr	SPort	DPort	Protocol	Transition	Action
r ₁	1.2.3.4	5.6.7.8	1080	80	P	Q ₀ + E ₂	ACCEPT
r ₂	5.6.7.8	1.2.3.4	80	1080	P	Q ₂ + E ₃	ACCEPT
r ₃	5.6.7.8	1.2.3.4	80	1080	P	Q ₅ + E ₇	ACCEPT
r ₄	1.2.3.4	5.6.7.8	1080	80	P	Q ₉ + E ₈	ACCEPT
r ₅	any	any	any	any	any	any	DENY

(a) Initial (flawed) rule set R

S	01	02	03	04	05	06	07	08
01	Q ₀ + E ₁	Q ₁ + E ₂	Q ₂ + E ₃	Q ₄ + E ₆	Q ₅ + E ₇	Q ₆ + E ₈	Q ₈ + E ₅	Q ₁₁ + E ₆
02	Q ₀ + E ₁	Q ₁ + E ₂	Q ₂ + E ₃	Q ₄ + E ₆	Q ₅ + E ₇	Q ₆ + E ₆	Q ₉ + E ₈	
03	Q ₀ + E ₁	Q ₁ + E ₂	Q ₂ + E ₃	Q ₄ + E ₆	Q ₅ + E ₈	Q ₇ + E ₅	Q ₁₀ + E ₇	
04	Q ₀ + E ₁	Q ₁ + E ₄	Q ₃ + E ₅	Q ₅ + E ₇	Q ₆ + E ₈	Q ₈ + E ₅	Q ₁₁ + E ₆	
05	Q ₀ + E ₁	Q ₁ + E ₄	Q ₃ + E ₅	Q ₅ + E ₇	Q ₆ + E ₆	Q ₉ + E ₈		
06	Q ₀ + E ₁	Q ₁ + E ₄	Q ₃ + E ₅	Q ₅ + E ₈	Q ₇ + E ₅	Q ₁₀ + E ₇		
07	Q ₀ + E ₂	Q ₂ + E ₃	Q ₄ + E ₆	Q ₅ + E ₇	Q ₆ + E ₈	Q ₈ + E ₅		
08	Q ₀ + E ₂	Q ₂ + E ₃	Q ₄ + E ₆	Q ₅ + E ₇	Q ₆ + E ₆	Q ₉ + E ₈		
09	Q ₀ + E ₂	Q ₂ + E ₃	Q ₄ + E ₆	Q ₅ + E ₈	Q ₇ + E ₅	Q ₁₀ + E ₇		

(b) Contents of set S

L	01	02	03	04	05	06	07
07	Q ₀ + E ₂	Q ₂ + E ₃	Q ₄ + E ₆	Q ₅ + E ₇	Q ₆ + E ₈	Q ₈ + E ₅	Q ₁₁ + E ₆
08	Q ₀ + E ₂	Q ₂ + E ₃	Q ₄ + E ₆	Q ₅ + E ₇	Q ₆ + E ₆	Q ₉ + E ₈	
09	Q ₀ + E ₂	Q ₂ + E ₃	Q ₄ + E ₆	Q ₅ + E ₈	Q ₇ + E ₅	Q ₁₀ + E ₇	

(c) Contents of set L

C	01	02	03	04	05	06	07
01	r ₁ : Q ₀ + E ₂	r ₂ : Q ₂ + E ₃	m ₁ : Q ₄ + E ₆	r ₃ : Q ₅ + E ₇	m ₃ : Q ₆ + E ₈	m ₆ : Q ₈ + E ₅	m ₈ : Q ₁₁ + E ₆
02	r ₁ : Q ₀ + E ₂	r ₂ : Q ₂ + E ₃	m ₁ : Q ₄ + E ₆	r ₃ : Q ₅ + E ₇	m ₄ : Q ₆ + E ₆	r ₄ : Q ₉ + E ₈	
03	r ₁ : Q ₀ + E ₂	r ₂ : Q ₂ + E ₃	m ₁ : Q ₄ + E ₆	m ₂ : Q ₅ + E ₈	m ₅ : Q ₇ + E ₅	m ₇ : Q ₁₀ + E ₇	

(d) Contents of set C

Rule	SrcAddr	DstAddr	SPort	DPort	Protocol	Transition	Action
r ₁	1.2.3.4	5.6.7.8	1080	80	P	Q ₀ + E ₂	ACCEPT
r ₂	5.6.7.8	1.2.3.4	80	1080	P	Q ₂ + E ₃	ACCEPT
m ₁	1.2.3.4	5.6.7.8	1080	80	P	Q ₄ + E ₆	ACCEPT
r ₃	5.6.7.8	1.2.3.4	80	1080	P	Q ₅ + E ₇	ACCEPT
m ₄	1.2.3.4	5.6.7.8	1080	80	P	Q ₆ + E ₆	ACCEPT
r ₄	1.2.3.4	5.6.7.8	1080	80	P	Q ₉ + E ₈	ACCEPT
r ₅	any	any	any	any	any	any	DENY

(e) Sample updated rule set

Fig. 3 – Applying Algorithm 1 to a sample rule set with a closed policy.

unmarked at the beginning of the process. The first time Line 9 is executed rule r_1 is marked as visited and starts the process. Given that its *Action* attribute contains the value ACCEPT, a subset is built in Line 11 based on function `prune_paths` and the *Transition* attribute of r_1 (i.e., $Q_0 + E_2$). As a result, function `prune_paths` removes out from S all those paths not containing the transition $Q_0 + E_2$. The result is assigned to L . Fig. 3(c) shows a sample interpretation of L over a two-dimensional vector.

Based on L , R , and the attributes of r_1 , the algorithm calls in Line 15 to function `cover_with_rules`. This function provides a mapping between the sets of transitions in L , and those in rules of R that are consistent with the attributes of r_1 . In other words, it provides a correlation of rules in R that are consistent with r_1 and necessary to cover the transitions in L . All those rules being correlated during the process, are marked as visited, and indexed to the transitions they cover. In the end, those transitions in L not covered by any rule in R are also

marked, and a series of missing rules (consistent with the attributes of r_1) are generated and indexed as well. All this information is returned by function `cover_with_rules` and assigned to C . Fig. 3(d) shows a sample interpretation of the information returned by the process. We can see in our example that all the rules in R are partially covering the three paths in L . The first path is covered by rules r_1 , r_2 , and r_3 ; and requires four extra rules to be fully covered. First, rule m_1 is generated, so that its *Action* and *Protocol* attributes are equivalent to the one of r_1 (ACCEPT and P), and its remainder attributes (*SrcAddr*, *DstAddr*, *SPort*, *DPort*) are consistent with those of r_2 and r_3 (i.e., inverting their source and destination sense). Same rationale applies for the generation of rules m_2 to m_8 .

The final step is conducted in Line 16, in which function `extract_missing_rules` simply pulls out from C one series of missing rules. In our example, we assume that the function returns just one of the paths in C (e.g., the shortest and

most covered path). In most cases, this solution is enough to signal the discovered misconfiguration and guide the security administrator on the correction of the rule set. This way, we consider that function `extract_missing_rules` derives from C rules m_1 and m_4 , and signal as possible correction the rule set shown in Fig. 3(e). However, as we will discuss in Section 3.5, straightforward modifications of Algorithm 1 and the aforementioned function can be done to fulfil other strategies (e.g., report the three rule sets that could be derived from set C in Fig. 3(d)). In any case, the information is reported in Line 18 to the security administrator, who is in charge of taking the final decision. Finally, and since all the rules in R have been marked as visited during the construction of C in Line 15 (the last rule signalling the default policy does not count), the condition in Line 20 holds and the verification process ends.

3.4. Complexity of the algorithm

The space consumption complexity of Algorithm 1 is bounded by Functions `find_all_paths` and `transitions_to_state`. The problem to solve by Function `find_all_paths` can efficiently be accomplished with either depth-first or breadth-first search of the automaton provided as input, so that the process recursively computes and returns all the paths from the initial to the final state of the automaton. Therefore, its space consumption complexity is linear in the total length of all the paths (at most, c times the number of paths, where c is a constant). An example is shown in Fig. 3(b). Function `transitions_to_state` simply returns all those terminal transitions associated with a given state of the automaton (the invalid state). Therefore, its space consumption complexity is linear in the total number of invalid transitions. An example is shown in Appendix A, Fig. 9. The resulting structures are computed just once at the beginning of the algorithm, regardless the number of iterations or length of the rule set.

The time complexity of Algorithm 1 is bounded by the complexity of Function `cover_with_rules`. As we have seen in the previous section, the problem to solve by this function is a special case of the set covering problem (Balas and Padberg, 1972). Therefore, its complexity is NP-complete: (1) the problem is NP since checking the validity of a solution, i.e., comparing the set of transitions covered by rules in a rule set, can be done in polynomial time; (2) there exists another known NP-complete problem, any instance of which can be reduced in polynomial time to an instance of our problem. Given that the size of protocol automata containing valid and invalid transitions is expected to be rather small, it is reasonable to use any $O(n^{\Omega(\log \log n)})$ -time deterministic function based on the simple greedy polynomial time heuristic defined in Chvatal (1979). Moreover, notice that within the main loop of the algorithm, Function `cover_with_rules()` is not necessarily computed for every rule in the set provided as input. All those rules correlated with the one being inspected are also marked as visited during the execution of the coverage function, so that the number of iterations is not greater than necessary. In the experimentations we have done (cf. Section 5, Fig. 6), we noticed that this significantly reduces the processing time consumption.

3.5. Discussion

During the processing of a rule set with a closed policy, Algorithm 1 verifies only that, at least, one of the paths to progress from Q_0 to Q_n via valid states of the automaton is fulfilled. Notice that the analysis technique could also verify that all possible paths are covered, i.e., verifying redundant stateful rules covering the automaton as a whole. We consider that such a redundant property of the analysis does not reflect the regular practises followed when configuring a given stateful firewall. However, if that would be necessary, the modification of the algorithm is straightforward and does not change its complexity. It simply relies on affecting set S to L in Line 11, instead of `prune_paths(S,ri[Transition])`; and instructing to function `extract_missing_rules` in Line 16 to extract all missing rules, instead of those covering only one of the paths. On the contrary, Algorithm 1 processing a rule set with an open policy verifies that all possible transitions that end in the invalid state are fulfilled. The opposite, i.e., verifying only one transition case per state (i.e., from valid to invalid state), would not be correct.

A second observation is about the case where the initial rule set contains permissions associated with invalid transitions, or prohibitions associated with valid transitions. Most scenarios applying to this case are already detected and corrected by the algorithms in Cuppens et al. (2012), as we have shown in Section 3.1. For instance, in the example shown in Table 1, we have the case of a prohibition associated with a valid transition that is detected and corrected once it is correlated to the remainder permissions in the set. However, the anomaly would certainly not be handled if such other explicit rules were not in the audited rule set. Although, in our opinion, these scenarios seem artificial, straightforward modifications of Algorithm 1 can be done to handle these cases as well, and complexity does not change. In the case of permissions associated with invalid transitions, the anomaly can be detected after the execution of Line 11, since the use of Function `prune_paths` with a rule containing a permission to the invalid state will return the empty set. Therefore, it suffices to add this constraint and report the anomaly to security administrator. In the second case, the anomaly can be detected in Line 15, by instructing to Function `cover_with_rules` to report the case in which none of the invalid transitions in the set provided to the function is covered by the rule being inspected.

In terms of formally proving the correctness of our approach, notice that the solution builds upon the series of transformations published in Garcia-Alfaro et al. (2008, 2006). Such transformations could alter existing configurations. Formal proofs required to prove their correctness, and guaranteeing that no anomalies are introduced, are available in Garcia-Alfaro et al. (2008, 2006) and citations thereof.

4. Inter-state rule misconfiguration

We have previously addressed the case of intra-state rule misconfiguration, where sets of stateful rules containing anomalies put in risk the inner logic of connection-oriented protocol states. The use of both stateful and stateless rules

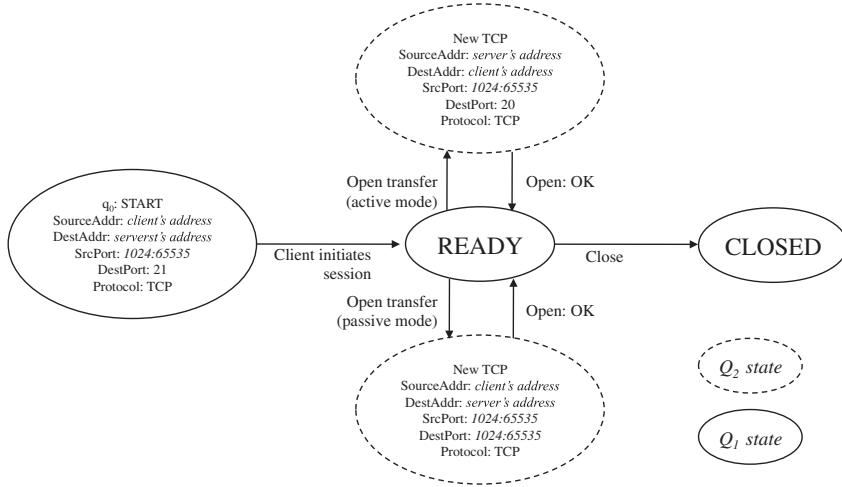


Fig. 4 – Suggested automaton for the application layer protocol FTP.

may be also found in a firewall configuration. For instance, a security administrator may add a rule in order to handle forwarding of connections which are used to transferring data in FTP sessions. On Linux based firewalls, we can manage this situation by adding a rule with the RELATED state parameter to the conntrack module. This rule will be inserted in the other stateless rules which have been previously defined in the rule set. We then search for anomalies between stateful and stateless rules based on the specification of a transport layer protocol. Most application layer protocols use several lower-layer protocol connections during a session between two nodes. This applies to FTP, IRC or VoIP protocols which use related transport-layer protocol connections. Let us further elaborate on the FTP case. A typical FTP session consists on the following two steps:

1. The client starts the session with the FTP server on port 21; a TCP connection — on the control plane — is established;
2. When the client wants to transfer data (file transfer, directory listing, etc.), two cases may occur:
 - Active mode: after the connection negotiation on the control plane, the server initiates a new transport layer connection for the data transfer, from the port 20 to a client's given port;
 - Passive mode: the data transfer connection is directly initiated from the client to an FTP server's given port.

The configuration of the firewall protecting the FTP server may contain:

- A stateless rule for allowing transport layer packets with the destination port 21;
- A stateful rule for allowing packets whose associated transport layer connection is marked with a related connection in an FTP session. The destination port will be either 20 (active mode) or greater than 1024 (passive mode).

In the previous example, one issue consists in correctly handling the related transport layer connections between two

nodes using an application layer protocol. First, the firewall should understand the given application layer protocol concerned by the rules in order to identify related connection packets. For instance, a Linux based stateful firewall can handle this case based on the RELATED state provided by the conntrack system, and can be added to those rules of the filtering table of iptables via the parameters `-m conntrack` and `-ctstate RELATED`. If such options are enabled in the firewall, the security administrator can then specify stateful rules to define the filtering rules in a much more granular way. If this is not done, it shall be reported as an inter-state misconfiguration anomaly. To automatically identify such anomalies for any given protocol, we also assume knowing the full specification of the connection-oriented protocol. This specification shall explain how connections are initiated and how related actions are triggered during a given session (order, number, ports, etc.). The first step consists in searching the stateless rules which stand for the establishment of the protocol connection. In the case of FTP, we search a rule which matches the transport layer packets with the destination port 21. If such rules are found, we consider the three following cases:

1. Stateful rules exist in the configuration to handle the possible related connections that may be used by the application layer protocol;
2. Stateless rules exist to handle these connexions;
3. No rule is defined to handle the related connexions.

The case 2 is too general because it does not take into account the inner logic of the application layer protocol. An attacker may be able to initiate a transport layer connection on a port which will be used only for a related connection of an application session. For example, an FTP connection on the server's port p will be allowed only if the server has previously initiated an FTP transfer on passive mode with a client on such a port p . In the case 3, the application session may fail because the firewall will probably deny the related connections. The case 1 solves the encountered problem with the other ones and complies with the protocol specification. In

a Linux firewall, such rules may be specified using the RELATED state.

4.1. Definitions

Our proposed audit process aims at assisting the security administrator to detect and fix cases 2 and 3 of the aforementioned anomaly. We first provide the data structures and functions that will be used to conduct the audit process:

- L : set of stateless rules, such that every rule L_i (where i is a natural integer) is characterised by the following conditions $L_i[\text{SrcAddr}]$, $L_i[\text{DstAddr}]$, $L_i[\text{SPort}]$, $L_i[\text{DPort}]$ and $L_i[\text{Protocol}]$ (such as TCP, UDP, or any other transport layer protocol).
- F : set of stateful rules, such that every rule F_i is characterised by the same conditions as the rules in L , plus the condition attribute $F_i[\text{State}]$. It is important to consider such a protocol, since the protocol of a given connection could be different from the protocol of the main connection. For instance, in scenarios based on VoIP applications, data transfer might be carried upon UDP traffic, while the main connection is relayed via TCP connections.
- A : deterministic finite automaton that describes an application layer protocol. We rely on the use of the alphabet of events and table of transitions of A , containing the set of operations that can be exchanged between hosts, e.g., remainder set of operations once the main connection of two FTP entities has been established. Q is the set of states, from which we identify the subset Q_2 . The elements q of Q_2 represent establishment of adjacent connections (such as TCP connections or from any other protocol type). The elements are characterised by the same set of conditions as the one in the rules (i.e., $q[\text{SrcAddr}]$, $q[\text{DstAddr}]$, etc.). Let us observe that $q[\text{State}]$ will highly rely on the specific firewall vendor (cf. following function definition, in which we define the way to link the specific state attribute of the automaton to the corresponding firewall device). Notice that $R_i[\text{State}]$ (i.e., the state defined in a given rule R_i) corresponds to the specific state as it is represented by the underlying firewall that contains the rule, not the state attribute of the automaton. If necessary, we can rely on extended features to provide a more fine-grained state management of some application layer protocols (Venkatamohan, 2011). $Q_1 = Q - Q_2$ contains the set of states that are independent from related connections, and for which the element $q[\text{State}]$ is not defined. Finally, the initial state q_0 of the automaton holds the following condition attributes: $q_0[\text{SPort}]$, $q_0[\text{DPort}]$ and $q_0[\text{Protocol}]$ (corresponding to the connection-oriented layer protocol). Fig. 4 depicts a sample automaton based on our construction, for the FTP protocol.

- $\text{state_firewall}(q)$: function that links a given state $q \in Q_2$ of the corresponding state automaton to the firewall. For instance, in the case of the FTP protocol and a Linux firewall based on `iptables` and `conntrack`, this function returns parameters `-m conntrack` and `-ctstate RELATED` for those states where the establishment of connections is called.
- $\text{rule_exists}(R, q)$: boolean function. R is a set of either stateless or stateful rules (but not both), q represents a state of the automaton A which belongs to Q_2 (the state corresponding to the establishment of related connections). If R contains stateless rules, then $\text{rule_exists}(R, q)$ is true only when there exists exactly one rule $R_i \in R$, such that $q[\text{SrcAddr}] \in R_i[\text{SrcAddr}]$, $q[\text{DstAddr}] \in R_i[\text{DstAddr}]$, $q[\text{SPort}] \in R_i[\text{SPort}]$, $q[\text{DPort}] \in R_i[\text{DPort}]$, and $q[\text{Protocol}] = R_i[\text{Protocol}]$. If R contains stateful rules, then $\text{rule_exists}(R, q)$ is true when the previous conditions also hold and, moreover, $\text{state_firewall}(q[\text{State}]) = R_i[\text{State}]$.
- $\text{rule_exists}(L, q_0)$: boolean function. q_0 contains the initial state of the protocol, and L is a set of stateless rules. The function is true only when there exists a rule $L_i \in L$, such that $q_0[\text{SPort}] \in L_i[\text{SPort}]$, $q_0[\text{DPort}] \in L_i[\text{DPort}]$, $q_0[\text{Protocol}] = L_i[\text{Protocol}]$.

4.2. Algorithms

Algorithm 2 enables the verification of every state Q_2 of an automaton associated with a given protocol, in order to find rules that can be correlated. The algorithm specifies the appropriate corrections in accordance to the detection of inter-state misconfiguration, and following the three cases mentioned above (absence of rules, or misconfigured stateless or stateful rules). **A**[Q_2] points out to the Q_2 set of the automaton. **Algorithm 3** allows detection and correction of inter-state misconfiguration between stateless and stateful rules, provided that a library of application layer protocols is given as input. Such a library must contain the corresponding automata for the protocols. Then, it verifies whether the firewall handles each of them, by looking at the initial state attribute q_0 of the corresponding automaton. In such a case, **Algorithm 2** processes the specific anomalies associated with that protocol. **A**[q_0] points out the initial state q_0 of every automaton. **Listing 3** presents an extract from a Linux based firewall configuration. It contains a closed policy with three stateless rules that aim at granting authorisation to node 1.2.3.4 for accessing the FTP service of node 5.6.7.8 (both in active and passive mode).

The sample shown in Listing 3 contains two rules affected by inter-state misconfiguration. Rule in Line 02 is a stateless authorisation to control incoming higher port connections targeting the FTP server listening on port 21. Then, rules in Lines 03 and 04 grant authorisation access to the data connection counterpart, i.e., outgoing connection from the server to the

```

01: iptables -P FORWARD DROP
02: iptables -A FORWARD -p tcp -s 1.2.3.4 -d 5.6.7.8 --sport 1024:65535 --dport 21 -j ACCEPT
03: iptables -A FORWARD -p tcp -s 5.6.7.8 -d 1.2.3.4 --sport 20 --dport 1024:65535 -j ACCEPT
04: iptables -A FORWARD -p tcp -s 1.2.3.4 -d 5.6.7.8 --sport 1024:65535 --dport 1024:65535 -j ACCEPT

```

Listing 3. Sample flawed configuration based on a Linux based firewall.

client. However, these last two rules are stateless. They grant access to any connection targeting ports in the whole range from 1024 to 65,535. If we apply [Algorithm 3](#) to the previous configuration, it will detect such a situation and suggest to handle the discovered issue by adding to Lines 03 and 04 the parameters `--m conntrack` and `--ctstate RELATED`.

```

1: Input:  $L$ , set of stateless rules
2: Input:  $F$ , set of stateful rules
3: Input:  $A$ , protocol automaton
   /*  $A[Q_2]$ :  $Q_2$  states for automaton  $A$  */
4: for all  $q \in A[Q_2]$  do
5:   if rule_exists( $F, q$ ) then
6:     /* Move to following state */
7:     continue;
8:   else if rule_exists( $L, q$ ) then
9:     report('stateless rule for state  $q$  of protocol  $A$ ')
10:   else
11:     report('missing rule for state  $q$  of protocol  $A$ ')
12:   end if
13: end for
```

Algorithm 2. `handle_inter_rule_misconfiguration(L, F, A)`.

a generic and reusable way. The result is a system that builds Platform-specific models (PSMs) from firewall configuration files (e.g., `iptables`-scripting files), and transforms them into a Platform-independent model (PIM). Then, we apply our discovery algorithms at the PIM level. This way, the solution obtained for a given firewall (e.g., a Linux firewall based on `iptables`) is reusable for any other system with little effort.

[Fig. 5](#) summarises the approach followed by our prototype to handle misconfigured files. It comprises the following steps: (1) Parsing and injection of already deployed configuration files into models at the PSM level; (2) Transformation of models from the PSM level to the PIM level; (3) Alignment of models at the PIM level w.r.t. a given protocol automaton; (4) Execution of misconfiguration algorithms and reporting of the discovered anomalies; (5) Generation of corrective rules. During the first phase, the prototype parses already deployed configuration files and inject their information into a PSM representation. This step constitutes a bridge between technical spaces allowing to pass from the technical space of configuration files (grammars and text files) to the technical space of the model-driven methodol-

```

1: Input:  $L$ , set of stateless rules
2: Input:  $F$ , set of stateful rules
3: Input:  $Library$  of automata, containing the list of supported application-layer protocols
4: for all  $A \in Library$  do
5:   if rule_exists( $L, A[q_0]$ ) then
6:     handle_inter_rule_misconfiguration( $L, F, A$ );
7:   end if
8: end for
```

Algorithm 3. `handle_all_protocols (L, F, Library)`.

5. Experimental results

In order to validate the feasibility of our approach, a proof-of-concept prototype has been developed under the `Eclipse` framework. The prototype, available at [E-git repository](#), also provides an extension of MIRAGE ([Garcia-Alfaro et al., 2011](#)), a firewall audit tool for the automatic detection and correction of stateless firewall configuration anomalies. The development of the stateful features is based on model-driven engineering, and extends the results presented in [Martinez et al. \(2012\)](#) for the stateless case. Model-driven engineering promotes the use of abstract software models, representing the concepts of a problem domain. In our case, this means extracting and verifying already deployed stateful firewall configuration scripts affected by rule misconfiguration. The goal of using model-driven engineering is getting rid off the low level details of the concrete solution system, so that we can focus on the problem itself. This has enabled us with the possibility of abstracting from the rule filtering language and providing the precise implementation of our algorithmic solutions in

ogy (metamodels and models) ([Kent, 2002](#)). To build the necessary parser for this step, we have used `Xtext`, an eclipse-based framework for building domain specific languages. A parser for `iptables` configurations has been implemented as follows. First, we have specified the grammar corresponding to the user-space `iptables` tool necessary to configure Linux firewalls. Then, the grammar has been used as an input for the `Xtext` framework. As a result, we have obtained the specific metamodel for `iptables` rules, as well as its parser and editor. Based on these three generated artifacts, our prototype can now inject existing configuration files into models at the PSM level (i.e., from existing Linux `iptables` configuration files to a PSM of `iptables` conforming the metamodel obtained via the `Xtext` framework). [Appendix B](#), [Fig. 11](#), shows a simplified version of the grammar we used to build the three artifacts. [Appendix B](#), [Fig. 12](#), shows the graphical representation of the metamodel that can be obtained by providing such a grammar to `Xtext`. This metamodel, as well as all the others described in this section, are implemented as an EMF `Eclipse Modeling Framework` Ecore model.

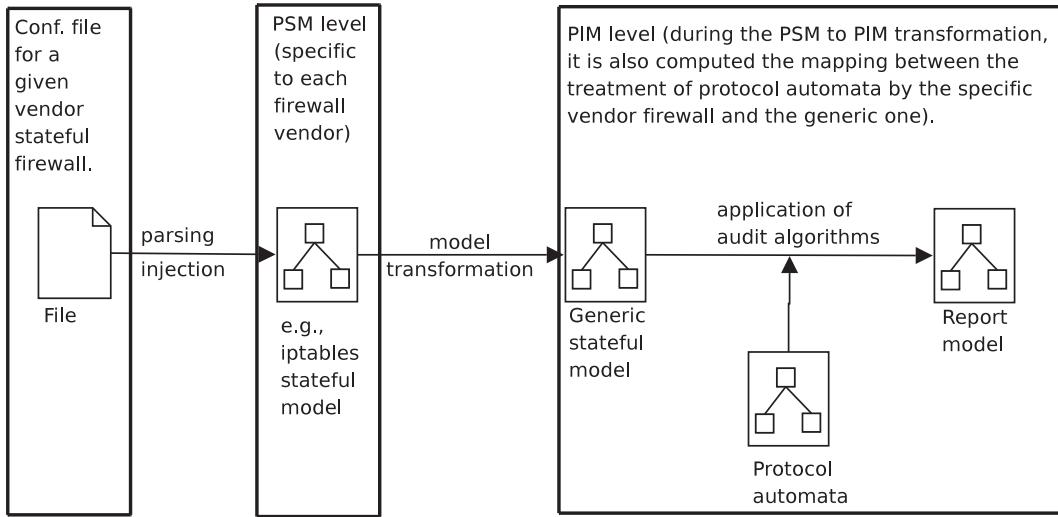


Fig. 5 – Our proposed model-driven evaluation framework.

Once the specific model at the PSM level is available, a transformation towards a generic stateful filter model at the PIM level is performed. The PIM metamodel that we use is an extension of the one presented in Martinez et al. (2012). Appendix B, Fig. 13, shows a sample graphical representation of our new metamodel. The previous metamodel contained only two entities: (1) Host, that represents network hosts as they are represented in the configuration files, i.e., as IP addresses and IP ranges; and (2) Connection, that represents connections between hosts specifying the port used to make the connection, the protocol (protocolKind in {icmp, tcp, udp}) and, if the connection, is allowed or denied (connectionKind in {ACCEPT, DENY}). That was a simplified representation of some relevant information contained in the configuration files of stateless packet-filter firewalls while eliminating the redundancy and readability problems that low level filter rule languages present. To make it able to represent stateful information as well, State and Event entities have been added. These two fields correspond to the Transition attribute shown in Section 3, for our generic stateful filter model. This way, State represents the state a connection is w.r.t. the finite state machine of a connection-oriented protocol (e.g., TCP, TCP, DCCP, ATM, Frame Relay, TIPC, SCTP, etc.); and Event represents the triggering condition for the transitions of such a state machine. For instance, with regard to the examples in Section 3, the Event field would be characterised by Flags and direction of a given protocol packet. The information to be injected in these two entities, as well as the other entities, comes from the original files parsed in the first step. For the model transformations, we use the ATL (Jouault and Kurtev, 2006) model-to-model transformation language. ATL is a hybrid (declarative with imperative facilities) language and framework that provides the means to easily specify the way to produce target models from source models. In Appendix B, Fig. 14, we show an ATL example that deals with the transformation of rules from the PSM level (e.g.,

iptables rules) to the PIM level (e.g., rules of the generic stateful filter model where we will apply the algorithms). The transformation keeps those general parameters (states, source and destination addresses, etc.) of every iptables rule while it gets rid of any unnecessary specific values (e.g., notion of tables, chains, etc.). During the transformation, it is also computed the mapping between the treatment of protocol automata by the specific vendor firewall and the generic one. Some more examples of ATL transformations of our prototype are available at [E-git repository](#).

The application of the audit algorithms presented in this paper is done at the PIM level. The algorithms themselves have been encoded as ATL transformations. This way, the application of the algorithms and functions is independent of the specific firewall. The output of the audit process is a new model that contains all the necessary feedback to handle the detected misconfiguration, such as the missing rules needed to handle it. At the time of writing this paper, a complete implementation of Algorithm 1, together with a PSM to PIM transformation based on stateful Linux firewall configuration files is available for our prototype at [E-git repository](#). A sample screenshot with the results of such an implementation is shown in Appendix B, Fig. 15. We show in the screenshot how a configuration file, based on iptables and the conntrack match, is processed. The output model is displayed in the console window, to guide the user on the necessary steps to update and correct an initial set with flawed rules. Using the framework, we conducted some tests and measured the memory consumption and the processing time needed to audit flawed configuration files based on the following two classes: (1) closed policies, containing only permissions to valid TCP transitions; (2) open policies, containing only prohibitions to invalid TCP transitions. The results of these measurements are plotted in Fig. 6(a) and (b). Notice that the plots are consistent with the complexity analysis discussed in Section 3.4 (in terms of space and time complexity).

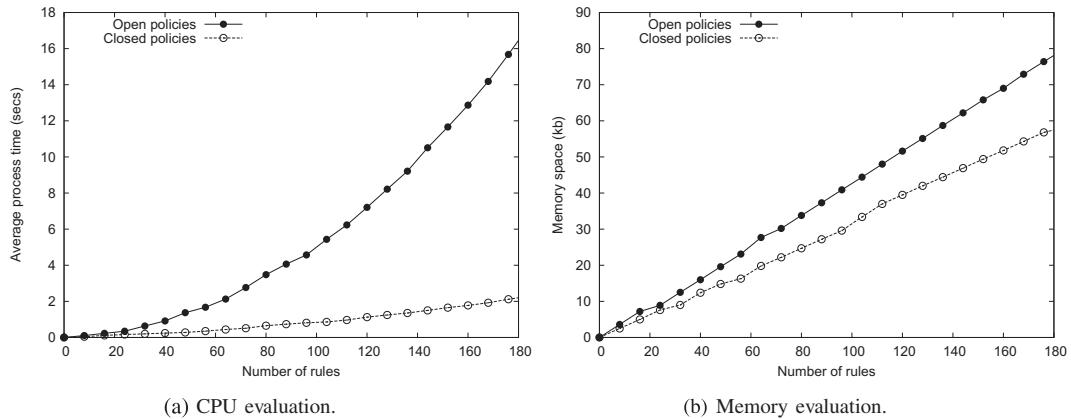


Fig. 6 – Experimental evaluations conducted using the framework shown in Fig. 5, and ATL transformations implementing Algorithm 1. The evaluations use incremental configuration files (from 8 to 180 rules), based on iptables and the conntrack match, representing (1) open policies, containing only prohibitions to invalid TCP transitions; and (2) closed policies, containing only permissions to valid TCP transitions. (a) Processing time needed to audit the files. (b) Space necessary to store the associated structures in memory.

Although the results show strong requirements, we believe that they are reasonable for off-line analysis, since the whole process does not affect the critical performance of the audited firewall.

6. Related work

Traditional research work on the design of firewalls, essentially stateless firewalls, mainly address the construction of high level languages for the specification of firewall configurations. This includes functional languages (Guttmann, 1997), rule-based languages (Bartal et al., 2004) and higher abstract models that allow capturing some further aspects such as network topologies (Cuppens et al., 2004). Such languages allow security administrators to free themselves from the technical complexity and specificity of proprietary firewall languages. Some of them allow, moreover, the automatic derivation of concrete access control rules to configure specific firewalls through a translation process. At the same time, research and development work in this context may allow the verification of consistency (i.e., absence of conflicts), completeness (all the expected requirements are covered), and compactness (none of the rules are redundant or unnecessary) (Preda et al., 2010). Refinement approaches may also take into account the functionality offered by every firewall manufacturer (stateless, stateful, management of virtual private networking, etc.) (Preda et al., 2011) to ensure the effective distribution of tasks between a decision module and the eventual filtering (enforcement) components.

For the already deployed firewall configurations, the aforementioned approaches do not solve redundancy or configuration conflicts that might have been introduced due to periodic, often manual, updates. Several studies have been conducted towards audit mechanisms that analyse already deployed configurations, with the goal of signalling inconsistencies and fixing the discovered anomalies. We

can classify them into three categories: (I) those that are oriented towards directly querying the firewall itself (Hazelhurst et al., 2000; Liu et al., 2004; Mayer et al., 2000), (II) those targeting conflict management (Babu and Varghese, 2001; Eppstein and Muthukrishnan, 2001) and (III) those focussing on the detection of anomalies (Al-Shaer and Hamed, 2003, 2004; Garcia-Alfaro et al., 2006, 2008, 2007). In category I, the analysis problem is relayed towards a process of information retrieval by directly querying the firewall. This requires having highly structured configurations and specific query languages for processing them, as well as for generating complete and effective data queries. The results are, moreover, prone to both false negatives and false positives, since no track from previous filtering matches are taken into account during the audit process. Category II is concerned with packet classification algorithms, mostly for packet filtering routers, and that rely on optimised data structures to speed up the matching process between incoming flows of packets and filtering rules. Then, the goal is to verify that there are no conflicting situations in which several rules with different actions (e.g., accept or reject the traffic) apply to the same traffic. Examples in this category include the use of techniques such as grid-of-tries classification (Srinivasan et al., 1999) and bit vector aggregation (Babu and Varghese, 2001). Class III improves the detection offered by solutions in class II, by: (1) characterising in more detail the set of anomalies, e.g., redundancy is also addressed; (2) transforming the rule sets in such a way that the ordering of rules is no longer relevant; (3) considering combinations of rules instead of simply comparing rules two by two as proposed by Al-Shaer and Hamed (2003), which enables the detection of a combination of rules that conflict with another rule (Garcia-Alfaro et al., 2006, 2008); and (4) extending the process to distributed setups with multi-firewall scenarios, in order to detect situations in which different firewalls within interconnected paths may perform different actions to the same network traffic.

None of the above surveyed techniques consider the case of stateful firewalls. So far, little work in the stateful case has been conducted. Butyan et al. (2009) proposed in an early approach that heads towards this research line. Built upon an existing tool reported by Yuan et al. (2006) in the solution is limited to the adaptation of existing anomaly detection techniques for stateless firewalls, such as redundancy and shadowing, to those that are stateful. Therefore, their work does not take into account anomalies that may impact, for instance, the tracking of connections or the management of the internal firewall state memory table. Automatic methods of theoretical nature based on high-level declarative languages (Gouda and Liu, 2005), theorem provers (Brucker and Wolff, 2007), and satisfiability solvers (Youssef and Bouhoula, 2011), have also been proposed as plausible solutions to conduct formal verification of stateful policies properties. The goal is to attest that a given firewall correctly implements the policy being verified. The practical application of these approaches to known (existing) firewall vendors is, however, unknown. In a different vein, Fitzgerald et al. (2008) propose in an approach based on semantic web technologies to model both stateless and stateful firewalls. Although the generality of their proposed representation is interesting enough, the work fails at characterising the precise types of errors that would be necessary to handle by the detection process in the stateful case. The approach only represents those good practises that must be followed when configuring a given stateful firewall.

7. Conclusion

Nowadays, packet filtering requires more than a passive solution to stop malicious traffic. Stateful firewalls are the predominant solution to guarantee such a protection. They provide an effective enforcement of access control rules at higher network layers, in order to protect incoming and outgoing interaction with the Internet. Nevertheless, the existence of anomalies in their configuration is very likely to degrade the protection they provide. While some anomalies may occur in rule sets that only contain stateful rules (*intra-state rule misconfiguration*), others affect rule sets that contain both stateful and stateless rules (*inter-state rule misconfiguration*). In this paper, we presented algorithmic solutions to handle anomalies for each of these two categories. Based on an automata theory approach, we provided solutions to detect inconsistent rules and report alternative configurations, in order to guide security administrators to handle such rules and get consistent rule sets. We validated the feasibility of our approach over a proof of concept prototype based on model-driven engineering. The model-driven engineering approach was chosen with the aim of separating the low level details of the problem (e.g., firewall vendor specificities) from the enforcement of the algorithmic solutions. This way, we applied the algorithms to the generic representation of the stateful filtering rules, at the abstract level. As a result, we expect to extend the prototype to address any other stateful filtering systems with minimum effort. Perspectives for further work

include the extension of our approach to handle inter-state rule misconfiguration and multi-component scenarios. In multi-component scenarios, several network security components are in charge of enforcing distributed network security policies, and would require a verification of the security functions supplied to them, to avoid the cases of misconfiguration reported in our work.

Acknowledgements

This research was partially supported by the European Commission, in the framework of the ITEA2 Predykot project (Grant agreement no. 10035). We also acknowledge support from the Spanish Ministry of Science and Innovation (grants TSI2007-65406-C03-03 E-AEGIS, TIN2011-27076-C03-02 CO-PRIVACY, TIN2010-15764 N-KHRONOUS and CONSOLIDER INGENIO 2010 CSD2007-0004 ARES). The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. We thank as well Xavier Rimasson and Tarik Moataz for all their help on the previous version of this paper, and Pablo Neira Ayuso for his fruitful remarks on several sections of this paper.

Appendix A. Applying Algorithm 1 to an open policy

We show in this appendix an intra-state misconfiguration example over a rule set enforcing the prohibition of invalid transitions, i.e., the default policy is to accept all those packets not matching any given prohibition. The complete table of transitions containing the invalid transitions is shown in Fig. 7 and the rule set is summarised in Fig. 8(a). The example complements the one given in Section 3.3, Fig. 3, on the application of Algorithm 1 to a sample rule set with a closed policy. The main difference here is the coverage of transitions applied in Line 15 of Algorithm 1. Notice that, instead of covering paths of the automaton from Q_0 to Q_{12} , we shall only verify the coverage of transitions to the invalid state (\emptyset) from any other state of the automaton. Hence, the contents of set L in Line 15 corresponds to the one-dimensional vector T built in Line 7 (cf. Fig. 9). Another difference is the representation of the rules in Fig. 8(a) and (b). To reduce their size, we grouped all the transitions sharing the same state and event path direction. We use the notation $Q_x + \{E_a|E_b\}$ to represent transitions $Q_x + E_a$ and $Q_x + E_b$. The remainder rationale of the algorithm for this second example prevails. Indeed, the application of function `cover_with_rules` in Line 15 of L is triggered by rule r_1 (first rule obtained in Line 9). This suffices to mark as visited all the remainder rules in R, and propose the updated rule set shown in Fig. 8(b). Notice that the 15 missing rules signalled in such a Figure, correspond to the 67 rules (out of 114) denoted as missing in Fig. 10. The remainder 47 rules in Fig. 10, already in R, correspond to rules r_1 to r_{11} shown in Fig. 8(a). We recall that the last rule in the set is not processed by the algorithm, since it is just the default policy action.

State \ Event	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}
Q_0 (LISTEN)	Q_1	Q_2	\emptyset							
Q_1 (SYN_SENT)	\emptyset	Q_2	\emptyset	Q_3	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
Q_2 (SYN_RECEIVED)	\emptyset	\emptyset	Q_4	\emptyset						
Q_3 (SA_RECEIVED)	\emptyset	\emptyset	\emptyset	\emptyset	Q_5	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
Q_4 (SA_SENT)	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	Q_5	\emptyset	\emptyset	\emptyset	\emptyset
Q_5 (ESTABLISHED)	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	Q_6	Q_7	\emptyset	\emptyset
Q_6 (FIN_SENT)	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	Q_9	\emptyset	Q_8	\emptyset	\emptyset
Q_7 (FIN_RECEIVED)	\emptyset	\emptyset	\emptyset	\emptyset	Q_{10}	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
Q_8 (CLOSING)	\emptyset	\emptyset	\emptyset	\emptyset	Q_{11}	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
Q_9 (ACK_RECEIVED)	\emptyset	Q_{12}	\emptyset	\emptyset						
Q_{10} (ACK_SENT)	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	Q_{12}	\emptyset	\emptyset	\emptyset
Q_{11} (LAST_ACK)	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	Q_{12}	\emptyset	\emptyset	\emptyset	\emptyset
Q_{12} (CLOSED)	\emptyset									

Fig. 7 – Complete set of transitions for the automaton depicted in Section 3, Fig. 2.

Rule	SrcAddr	DstAddr	SPort	DPort	Protocol	Transition	Action
r ₁	1.2.3.4	5.6.7.8	1080	80	P	Q ₀ + {E ₄ E ₆ E ₈ E ₁₀ }	DENY
r ₂	5.6.7.8	1.2.3.4	80	1080	P	Q ₀ + {E ₃ E ₅ E ₇ E ₉ }	DENY
r ₃	1.2.3.4	5.6.7.8	1080	80	P	Q ₁ + {E ₆ E ₈ E ₁₀ }	DENY
r ₄	5.6.7.8	1.2.3.4	80	1080	P	Q ₁ + {E ₁ E ₃ E ₅ E ₇ E ₉ }	DENY
r ₅	1.2.3.4	5.6.7.8	1080	80	P	Q ₂ + {E ₂ E ₄ E ₆ E ₈ E ₁₀ }	DENY
r ₆	5.6.7.8	1.2.3.4	80	1080	P	Q ₃ + {E ₁ E ₃ E ₇ E ₉ }	DENY
r ₇	1.2.3.4	5.6.7.8	1080	80	P	Q ₄ + {E ₂ E ₄ E ₈ E ₁₀ }	DENY
r ₈	1.2.3.4	5.6.7.8	1080	80	P	Q ₈ + {E ₂ E ₄ E ₆ E ₈ E ₁₀ }	DENY
r ₉	5.6.7.8	1.2.3.4	80	1080	P	Q ₈ + {E ₁ E ₃ E ₇ E ₉ }	DENY
r ₁₀	1.2.3.4	5.6.7.8	1080	80	P	Q ₁₁ + {E ₂ E ₄ E ₈ E ₁₀ }	DENY
r ₁₁	5.6.7.8	1.2.3.4	80	1080	P	Q ₁₁ + {E ₁ E ₃ E ₅ E ₇ E ₉ }	DENY
r ₁₂	any	any	any	any	any	any	ACCEPT

(a) Initial (flawed) rule set R

Rule	SrcAddr	DstAddr	SPort	DPort	Protocol	Transition	Action
r ₁	1.2.3.4	5.6.7.8	1080	80	P	Q ₀ + {E ₄ E ₆ E ₈ E ₁₀ }	DENY
r ₂	5.6.7.8	1.2.3.4	80	1080	P	Q ₀ + {E ₃ E ₅ E ₇ E ₉ }	DENY
r ₃	1.2.3.4	5.6.7.8	1080	80	P	Q ₁ + {E ₆ E ₈ E ₁₀ }	DENY
r ₄	5.6.7.8	1.2.3.4	80	1080	P	Q ₁ + {E ₁ E ₃ E ₅ E ₇ E ₉ }	DENY
r ₅	1.2.3.4	5.6.7.8	1080	80	P	Q ₂ + {E ₂ E ₄ E ₆ E ₈ E ₁₀ }	DENY
m ₁	5.6.7.8	1.2.3.4	80	1080	P	Q ₂ + {E ₁ E ₅ E ₇ E ₉ }	DENY
m ₂	1.2.3.4	5.6.7.8	1080	80	P	Q ₃ + {E ₂ E ₄ E ₆ E ₈ E ₁₀ }	DENY
r ₆	5.6.7.8	1.2.3.4	80	1080	P	Q ₃ + {E ₁ E ₃ E ₇ E ₉ }	DENY
r ₇	1.2.3.4	5.6.7.8	1080	80	P	Q ₄ + {E ₂ E ₄ E ₈ E ₁₀ }	DENY
m ₃	5.6.7.8	1.2.3.4	80	1080	P	Q ₄ + {E ₁ E ₃ E ₅ E ₇ E ₉ }	DENY
m ₄	1.2.3.4	5.6.7.8	1080	80	P	Q ₅ + {E ₂ E ₄ E ₆ E ₁₀ }	DENY
m ₅	5.6.7.8	1.2.3.4	80	1080	P	Q ₅ + {E ₁ E ₃ E ₅ E ₉ }	DENY
m ₆	1.2.3.4	5.6.7.8	1080	80	P	Q ₆ + {E ₂ E ₄ E ₁₀ }	DENY
m ₇	5.6.7.8	1.2.3.4	80	1080	P	Q ₆ + {E ₁ E ₃ E ₅ E ₇ E ₉ }	DENY
m ₈	1.2.3.4	5.6.7.8	1080	80	P	Q ₇ + {E ₂ E ₄ E ₆ E ₈ E ₁₀ }	DENY
m ₉	5.6.7.8	1.2.3.4	80	1080	P	Q ₇ + {E ₁ E ₃ E ₇ E ₉ }	DENY
r ₈	1.2.3.4	5.6.7.8	1080	80	P	Q ₈ + {E ₂ E ₄ E ₆ E ₈ E ₁₀ }	DENY
r ₉	5.6.7.8	1.2.3.4	80	1080	P	Q ₈ + {E ₁ E ₃ E ₇ E ₉ }	DENY
m ₁₀	1.2.3.4	5.6.7.8	1080	80	P	Q ₉ + {E ₂ E ₄ E ₁₀ }	DENY
m ₁₁	5.6.7.8	1.2.3.4	80	1080	P	Q ₉ + {E ₁ E ₃ E ₅ E ₇ E ₉ }	DENY
m ₁₂	1.2.3.4	5.6.7.8	1080	80	P	Q ₁₀ + {E ₂ E ₄ E ₆ E ₈ E ₁₀ }	DENY
m ₁₃	5.6.7.8	1.2.3.4	80	1080	P	Q ₁₀ + {E ₁ E ₃ E ₅ E ₉ }	DENY
r ₁₀	1.2.3.4	5.6.7.8	1080	80	P	Q ₁₁ + {E ₂ E ₄ E ₈ E ₁₀ }	DENY
r ₁₁	5.6.7.8	1.2.3.4	80	1080	P	Q ₁₁ + {E ₁ E ₃ E ₅ E ₇ E ₉ }	DENY
m ₁₄	1.2.3.4	5.6.7.8	1080	80	P	Q ₁₂ + {E ₂ E ₄ E ₆ E ₈ E ₁₀ }	DENY
m ₁₅	5.6.7.8	1.2.3.4	80	1080	P	Q ₁₂ + {E ₁ E ₃ E ₅ E ₇ E ₉ }	DENY
r ₁₂	any	any	any	any	any	any	ACCEPT

(b) Sample updated rule set

Fig. 8 – Applying Algorithm 1 to a sample rule set enforcing an open policy. For simplicity, we group transitions that share the same state and event direction path (w.r.t. the event description shown in Section 3.3, Fig. 2). This way, $Q_x + \{E_a|E_b\}$ represents transitions $Q_x + E_a$ and $Q_x + E_b$.

T	01	02	03	04	05	06
01	$Q_0 + E_4$	$Q_0 + E_6$	$Q_0 + E_8$	$Q_0 + E_{10}$	$Q_0 + E_3$	$Q_0 + E_5$
T	07	08	09	10	11	12
01	$Q_0 + E_7$	$Q_0 + E_9$	$Q_1 + E_6$	$Q_1 + E_8$	$Q_1 + E_{10}$	$Q_1 + E_1$
T	13	14	15	16	17	18
01	$Q_1 + E_3$	$Q_1 + E_5$	$Q_1 + E_7$	$Q_1 + E_9$	$Q_2 + E_2$	$Q_2 + E_4$
T	19	20	21	22	23	24
01	$Q_2 + E_6$	$Q_2 + E_8$	$Q_2 + E_{10}$	$Q_2 + E_1$	$Q_2 + E_5$	$Q_2 + E_7$
T	25	26	27	28	29	30
01	$Q_2 + E_9$	$Q_3 + E_2$	$Q_3 + E_4$	$Q_3 + E_6$	$Q_3 + E_8$	$Q_3 + E_{10}$
T	31	32	33	34	35	36
01	$Q_3 + E_1$	$Q_3 + E_3$	$Q_3 + E_7$	$Q_3 + E_9$	$Q_4 + E_2$	$Q_4 + E_4$
T	37	38	39	40	41	42
01	$Q_4 + E_8$	$Q_4 + E_{10}$	$Q_4 + E_1$	$Q_4 + E_3$	$Q_4 + E_5$	$Q_4 + E_7$
T	43	44	45	46	47	48
01	$Q_4 + E_9$	$Q_5 + E_2$	$Q_5 + E_4$	$Q_5 + E_6$	$Q_5 + E_{10}$	$Q_5 + E_1$
T	49	50	51	52	53	54
01	$Q_5 + E_3$	$Q_5 + E_5$	$Q_5 + E_9$	$Q_6 + E_2$	$Q_6 + E_4$	$Q_6 + E_{10}$
T	55	56	57	58	59	60
01	$Q_6 + E_1$	$Q_6 + E_3$	$Q_6 + E_5$	$Q_6 + E_7$	$Q_6 + E_9$	$Q_7 + E_2$
T	61	62	63	64	65	66
01	$Q_7 + E_4$	$Q_7 + E_6$	$Q_7 + E_8$	$Q_7 + E_{10}$	$Q_7 + E_1$	$Q_7 + E_3$
T	67	68	69	70	71	72
01	$Q_7 + E_7$	$Q_7 + E_9$	$Q_8 + E_2$	$Q_8 + E_4$	$Q_8 + E_6$	$Q_8 + E_8$
T	73	74	75	76	77	78
01	$Q_8 + E_{10}$	$Q_8 + E_1$	$Q_8 + E_3$	$Q_8 + E_7$	$Q_8 + E_9$	$Q_9 + E_2$
T	79	80	81	82	83	84
01	$Q_9 + E_4$	$Q_9 + E_6$	$Q_9 + E_{10}$	$Q_9 E_1$	$Q_9 E_3$	$Q_9 E_5$
T	85	86	87	88	89	90
01	$Q_9 E_7$	$Q_9 E_9$	$Q_{10} + E_2$	$Q_{10} + E_4$	$Q_{10} + E_6$	$Q_{10} + E_8$
T	91	92	93	94	95	96
01	$Q_{10} + E_{10}$	$Q_{10} + E_1$	$Q_{10} + E_3$	$Q_{10} + E_5$	$Q_{10} + E_9$	$Q_{11} + E_2$
T	97	98	99	100	101	102
01	$Q_{11} + E_4$	$Q_{11} + E_8$	$Q_{11} + E_{10}$	$Q_{11} + E_1$	$Q_{11} + E_3$	$Q_{11} + E_5$
T	103	104	105	106	107	108
01	$Q_{11} + E_7$	$Q_{11} + E_9$	$Q_{12} + E_2$	$Q_{12} + E_4$	$Q_{12} + E_6$	$Q_{12} + E_8$
T	109	110	111	112	113	114
01	$Q_{12} + E_{10}$	$Q_{12} + E_1$	$Q_{12} + E_3$	$Q_{12} + E_5$	$Q_{12} + E_7$	$Q_{12} + E_9$

Fig. 9 – Contents of set T , during the execution of Algorithm 1 with the automaton in Fig. 2.

<i>C</i>	01	02	03	04	05	06
01	r ₁ : Q ₀ + E ₄	r ₁ : Q ₀ + E ₆	r ₁ : Q ₀ + E ₈	r ₁ : Q ₀ + E ₁₀	r ₂ : Q ₀ + E ₃	r ₂ : Q ₀ + E ₅
<i>C</i>	07	08	09	10	11	12
01	r ₂ : Q ₀ + E ₇	r ₂ : Q ₀ + E ₉	r ₃ : Q ₁ + E ₆	r ₃ : Q ₁ + E ₈	r ₃ : Q ₁ + E ₁₀	r ₄ : Q ₁ + E ₁
<i>C</i>	13	14	15	16	17	18
01	r ₄ : Q ₁ + E ₃	r ₄ : Q ₁ + E ₅	r ₄ : Q ₁ + E ₇	r ₄ : Q ₁ + E ₉	r ₅ : Q ₂ + E ₂	r ₅ : Q ₂ + E ₄
<i>C</i>	19	20	21	22	23	24
01	r ₅ : Q ₂ + E ₆	r ₅ : Q ₂ + E ₈	r ₅ : Q ₂ + E ₁₀	m ₁ : Q ₂ + E ₁	m ₁ : Q ₂ + E ₅	m ₁ : Q ₂ + E ₇
<i>C</i>	25	26	27	28	29	30
01	m ₁ : Q ₂ + E ₉	m ₂ : Q ₃ + E ₂	m ₂ : Q ₃ + E ₄	m ₂ : Q ₃ + E ₆	m ₂ : Q ₃ + E ₈	m ₂ : Q ₃ + E ₁₀
<i>C</i>	31	32	33	34	35	36
01	r ₆ : Q ₃ + E ₁	r ₆ : Q ₃ + E ₃	r ₆ : Q ₃ + E ₇	r ₆ : Q ₃ + E ₉	r ₇ : Q ₄ + E ₂	r ₇ : Q ₄ + E ₄
<i>C</i>	37	38	39	40	41	42
01	r ₇ : Q ₄ + E ₈	r ₇ : Q ₄ + E ₁₀	m ₃ : Q ₄ + E ₁	m ₃ : Q ₄ + E ₃	m ₃ : Q ₄ + E ₅	m ₃ : Q ₄ + E ₇
<i>C</i>	43	44	45	46	47	48
01	m ₃ : Q ₄ + E ₉	m ₄ : Q ₅ + E ₂	m ₄ : Q ₅ + E ₄	m ₄ : Q ₅ + E ₆	m ₄ : Q ₅ + E ₁₀	m ₅ : Q ₅ + E ₁
<i>C</i>	49	50	51	52	53	54
01	m ₅ : Q ₅ + E ₃	m ₅ : Q ₅ + E ₅	m ₅ : Q ₅ + E ₉	m ₆ : Q ₆ + E ₂	m ₆ : Q ₆ + E ₄	m ₆ : Q ₆ + E ₁₀
<i>C</i>	55	56	57	58	59	60
01	m ₇ : Q ₆ + E ₁	m ₇ : Q ₆ + E ₃	m ₇ : Q ₆ + E ₅	m ₇ : Q ₆ + E ₇	m ₇ : Q ₆ + E ₉	m ₈ : Q ₇ + E ₂
<i>C</i>	61	62	63	64	65	66
01	m ₈ : Q ₇ + E ₄	m ₈ : Q ₇ + E ₆	m ₈ : Q ₇ + E ₈	m ₈ : Q ₇ + E ₁₀	m ₉ : Q ₇ + E ₁	m ₉ : Q ₇ + E ₃
<i>C</i>	67	68	69	70	71	72
01	m ₉ : Q ₇ + E ₇	m ₉ : Q ₇ + E ₉	r ₈ : Q ₈ + E ₂	r ₈ : Q ₈ + E ₄	r ₈ : Q ₈ + E ₆	r ₈ : Q ₈ + E ₈
<i>C</i>	73	74	75	76	77	78
01	r ₈ : Q ₈ + E ₁₀	r ₉ : Q ₈ + E ₁	r ₉ : Q ₈ + E ₃	r ₉ : Q ₈ + E ₇	r ₉ : Q ₈ + E ₉	m ₁₀ : Q ₉ + E ₂
<i>C</i>	79	80	81	82	83	84
01	m ₁₀ : Q ₉ + E ₄	m ₁₀ : Q ₉ + E ₆	m ₁₀ : Q ₉ + E ₁₀	m ₁₁ : Q ₉ E ₁	m ₁₁ : Q ₉ E ₃	m ₁₁ : Q ₉ E ₅
<i>C</i>	85	86	87	88	89	90
01	m ₁₁ : Q ₉ E ₇	m ₁₁ : Q ₉ E ₉	m ₁₂ : Q ₁₀ + E ₂	m ₁₂ : Q ₁₀ + E ₄	m ₁₂ : Q ₁₀ + E ₆	m ₁₂ : Q ₁₀ + E ₈
<i>C</i>	91	92	93	94	95	96
01	m ₁₂ : Q ₁₀ + E ₁₀	m ₁₃ : Q ₁₀ + E ₁	m ₁₃ : Q ₁₀ + E ₃	m ₁₃ : Q ₁₀ + E ₅	m ₁₃ : Q ₁₀ + E ₉	r ₁₀ : Q ₁₁ + E ₂
<i>C</i>	97	98	99	100	101	102
01	r ₁₀ : Q ₁₁ + E ₄	r ₁₀ : Q ₁₁ + E ₈	r ₁₀ : Q ₁₁ + E ₁₀	r ₁₁ : Q ₁₁ + E ₁	r ₁₁ : Q ₁₁ + E ₃	r ₁₁ : Q ₁₁ + E ₅
<i>C</i>	103	104	105	106	107	108
01	r ₁₁ : Q ₁₁ + E ₇	r ₁₁ : Q ₁₁ + E ₉	m ₁₄ : Q ₁₂ + E ₂	m ₁₄ : Q ₁₂ + E ₄	m ₁₄ : Q ₁₂ + E ₆	m ₁₄ : Q ₁₂ + E ₈
<i>C</i>	109	110	111	112	113	114
01	m ₁₄ : Q ₁₂ + E ₁₀	m ₁₅ : Q ₁₂ + E ₁	m ₁₅ : Q ₁₂ + E ₃	m ₁₅ : Q ₁₂ + E ₅	m ₁₅ : Q ₁₂ + E ₇	m ₁₅ : Q ₁₂ + E ₉

Fig. 10 – Contents of set C, during the execution of Algorithm 1 on the initial rule set shown in Fig. 8(a).

Appendix B. Experimental results (additonal material)

```

Model:
  rules += Rule*;

Rule:
  declaration=ChainDeclaration | FilterDeclaration
;

FilterDeclaration:
  filter=FilteringSpec
;

FilteringSpec:
  FilterSpec
;

FilterSpec:
  'iptables' option=(-A | -D | -P)
  chain=Chain ('-s' ip=IPExpr)?
  ('-i' interface=Interface)? ('-d' ipDst=IPExpr)?
  ('-p' protocol=Protocol)?
  ('--sport' sourcePort=INT)?
  ('--dport' destinationPort=INT)? (neg='!')
  (syn='--syn')? ('-m' matches=Match)?
  ('--ctstate' states+=State (',' states+=State)*)?
  ('--ctdir' dir=Dir)?
  ('--ctstatus' status=Status)?
  ('--state' states+=State (',' states+=State)*)?
  ('--tcp-flags' examFlags+=TCPFlag
  (',' examFlags+=TCPFlag)* flags+=TCPFlag
  (',' flags+=TCPFlag)*)?
  ('-j' target=Target)?
  ('--log-prefix' lp=LP)?
;

Match:
  name=(conntrack|state)
;

TCPFlag:
  name=(Syn | Ack | Fin | Rst | Psh |
        Urg | All | None)
;

ChainDeclaration:
  'iptables' '-N' ChainName
;

IPExpr:
  ipByteExpr '.' ipByteExpr '.'
  ipByteExpr ',' ipByteExpr
  (IpRangeExpr)?
;

ipByteExpr:
  INT
;

IpRangeExpr:
  '/' INT
;

```

Fig. 11 – Simplified Iptables grammar example.

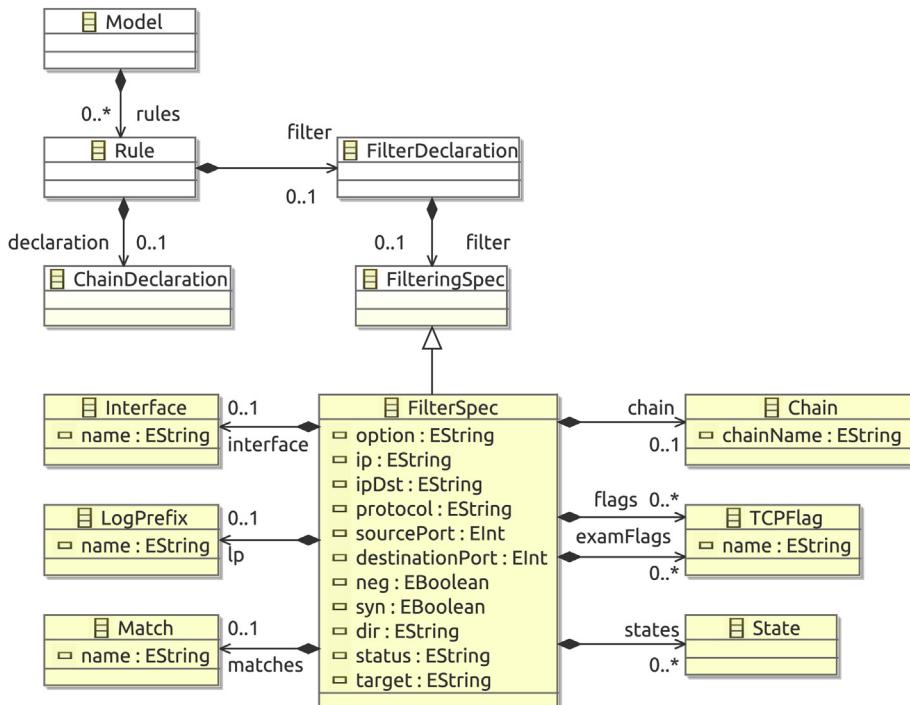


Fig. 12 – Iptables metamodel, obtained by applying the grammar in Fig. 11 on the Xtext framework.

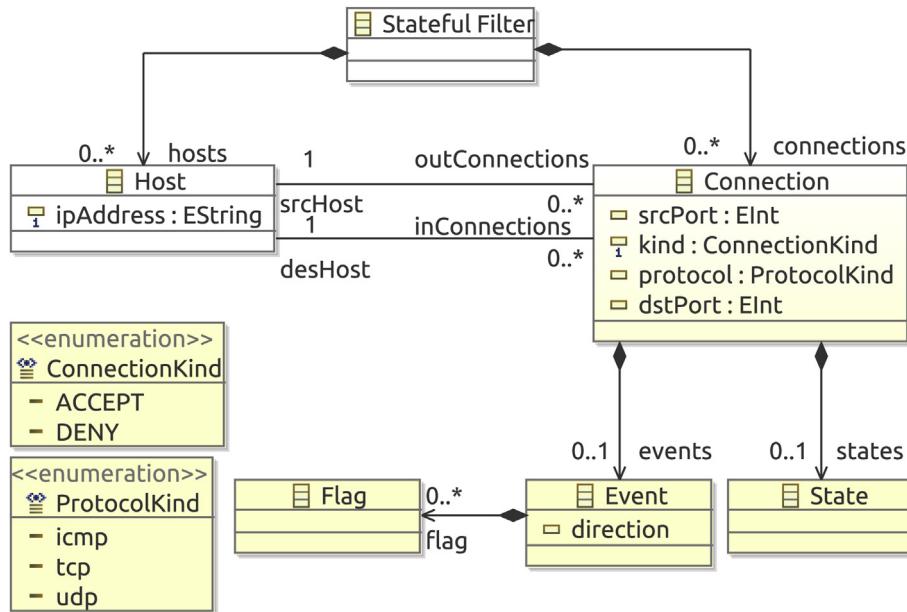


Fig. 13 – Stateful (generic) filter metamodel.

```

rule filterNew2ConnectionsQ0{
    from
        s1: Iptables!FilterSpec ,
        s2: Iptables!FilterSpec .(thisModule.FirstPathRules->includes(s1)
            and s1.isOfState('NEW', 'SYN')
            and s2.isOfState('NEW', 'SYN')
            and s2.ip = s1.ipDst
            and s2.ipDst = s1.ip)
    to
        t: StatefulPIM!Connection ->
        (StatefulPIM!Network.allInstancesFrom('OUT')->
        first().connections) (
            srcPort <- s1.sourcePort,
            dstPort <- s1.destinationPort,
            kind <- if s1.target = 'ACCEPT' then #ACCEPT else #DENY endif,
            protocol <- s1.protocol,
            states <- state,
            setFlags <- event,
            desHost <- thisModule.createHost(thisModule.findHost(s1.ipDst)),
            srcHost <- thisModule.createHost(thisModule.findHost(s1.ip)))
        ),
        state: StatefulPIM!State (
            name <- 'Q0'
        ),
        event: StatefulPIM!Flag (
            name <- 'E2'
        ),
        t2: StatefulPIM!Connection ->
        (StatefulPIM!Network.allInstancesFrom('OUT')->
        first().connections) (
            srcPort <- s2.sourcePort,
            dstPort <- s2.destinationPort,
            kind <- if s2.target = 'ACCEPT' then #ACCEPT else #DENY endif,
            protocol <- s2.protocol,
            states <- state2,
            setFlags <- event2,
            desHost <- thisModule.createHost(thisModule.findHost(s2.ipDst)),
            srcHost <- thisModule.createHost(thisModule.findHost(s2.ip)))
        ),
        state2: StatefulPIM!State (
            name <- 'Q0'
        ),
        event2: StatefulPIM!Flag (
            name <- 'E1'
        )
}

```

Fig. 14 – ATL transformation example.

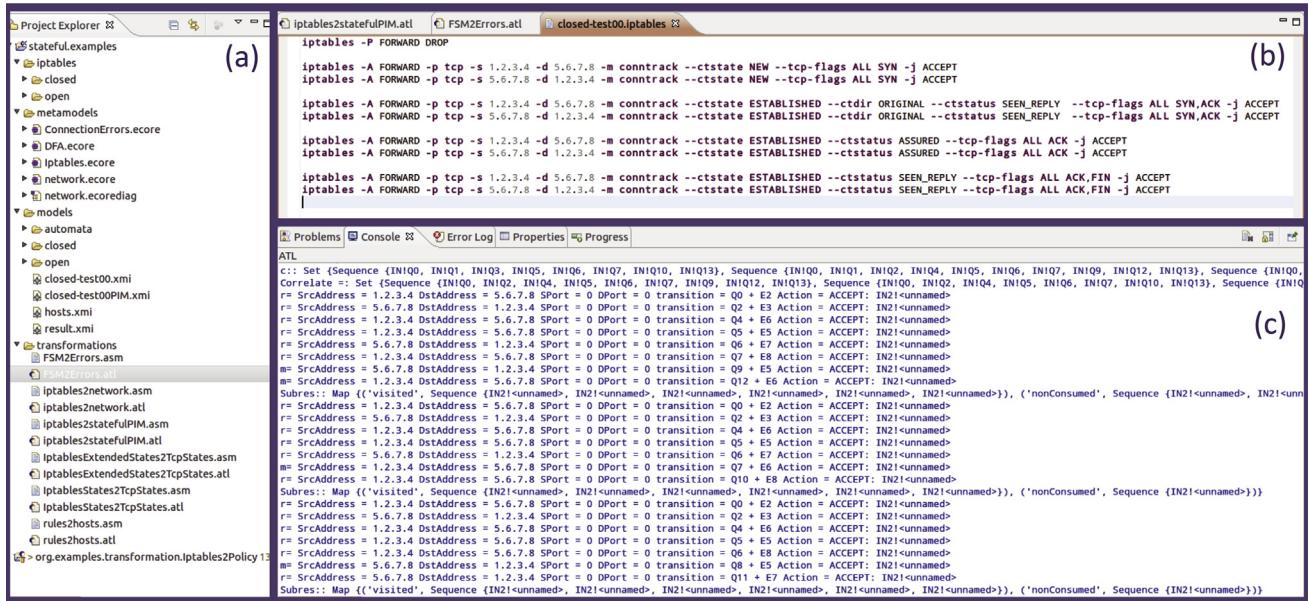


Fig. 15 – Prototype system developed under the Eclipse environment. (a) Project explorer, from which we can access to the configuration files, as well as models and transformations. (b) Initial rule set, affected by intra-state misconfiguration. (c) Console, displaying the results of the audit process and the series of missing rules to handle the misconfiguration.

REFERENCE

- Al-Shaer E, Hamed H. Firewall policy advisor for anomaly discovery and rule editing. In: Eighth international symposium on integrated network management; 2003. p. 17–30.
- Al-Shaer E, Hamed H. Discovery of policy anomalies in distributed firewalls. In: 23rd annual conference of the IEEE computer and communications societies (INFOCOM 2004); 2004. p. 2605–16.
- Arlitt M, Williamson C. An analysis of TCP reset behaviour on the internet. Computer Communication Review 2005;35(1):37–44.
- Baboscu F, Varghese G. Scalable packet classification. Computer Communication Review 2001;31(4):199–210.
- Balas E, Padberg M. On the set-covering problem. Operations Research 1972;20(6):1152–61.
- Bartal Y, Mayer A, Nissim K, Wool A. Firmato: a novel firewall management toolkit. ACM Trans. Comput. Syst 2004;22(4): 381–420.
- Brucker A, Wolff B. Test-sequence generation with Hol-TestGen with an application to firewall testing. Tests and Proofs 2007;149–68.
- Buttyan L, Pék G, Thong T. Consistency verification of stateful firewalls is not harder than the stateless case. Infocommunications Journal 2009;LXIV(1):2–8.
- Chvatal V. A greedy heuristic for the set-covering problem. Mathematics of Operations Research 1979;4(3):233–5.
- Cuppens F, Cuppens-Boulahia N, Sans T, Miège A. A formal approach to specify and deploy a network security policy. In: Formal aspects in security and trust (FAST'04); 2004. p. 203–18.
- Cuppens F, Cuppens-Boulahia N, Garcia-Alfaro J, Moataz T, Rimasson X. Handling stateful firewall anomalies. In: 27th IFIP international information security and privacy conference (SEC 2012); 2012. p. 174–86.
- MDE-MIRAGE git repository. URL: <https://github.com/mde-mirage/cose/>.
- Eclipse. The Eclipse Foundation open source community website. URL: <http://www.eclipse.org/>.
- Eclipse Modeling Framework Project. URL: <http://www.eclipse.org/modeling/emf/>.
- Eppstein D, Muthukrishnan S. Internet packet filter management and rectangle geometry. In: 12th Annual ACM-SIAM symposium on discrete algorithms. Society for Industrial and Applied Mathematics; 2001. p. 827–35.
- Fitzgerald W, Foley S, Foghlú M. Network access control interoperation using semantic web techniques. In: 6th international workshop on security in information systems (WOSIS); 2008. p. 26–37.
- Garcia-Alfaro J, Cuppens F, Cuppens-Boulahia N. Analysis of policy anomalies on distributed network security setups. In: 11th European symposium research computer security (ESORICS 2006); 2006. p. 496–511.
- Garcia-Alfaro J, Cuppens F, Cuppens-Boulahia N. Management of exceptions on access control policies. In: 22nd IFIP international information security and privacy conference (Sec 2012); 2007. p. 97–108.
- Garcia-Alfaro J, Boulahia-Cuppens N, Cuppens F. Complete analysis of configuration rules to guarantee reliable network security policies. International Journal of Information Security 2008;7(2):103–22.
- Garcia-Alfaro J, Cuppens F, Cuppens-Boulahia N, Preda S. MIRAGE: a management tool for the analysis and deployment of network security policies. In: 3rd international workshop on autonomous and spontaneous security (SETOP 2010). Springer; 2011. p. 203–15.
- Gouda M, Liu A. A model of stateful firewalls and its properties. In: 35th international conference on dependable systems and networks (DSN 2005); 2005. p. 128–37.
- Guttman J. Filtering postures: local enforcement for global policies. In: IEEE symposium on security and privacy; 1997. p. 120–9.
- Hari A, Suri S, Parulkar G. Detecting and resolving packet filter conflicts. In: 19th annual conference of the IEEE computer and communications societies (INFOCOM 2000), Vol. 3. IEEE; 2000. p. 1203–12.
- Hazelhurst S, Attar A, Sinnappan R. Algorithms for improving the dependability of firewall and filter rule lists. In: 30th

- international conference on dependable systems and networks (DSN 2000); 2000. p. 576–85.
- Jouault F, Kurtev I. Transforming models with ATL. In: Satellite events at the MODELS 2005 conference; 2006. p. 128–38.
- Kent S. Model driven engineering. In: Integrated formal methods. Springer; 2002. p. 286–98.
- Liu A, Gouda M, Ma H, Ngu A. Firewall queries. In: 8th international conference on principles of distributed systems; 2004. p. 124–39.
- Nmap 6. The Network Mapper. URL: <http://nmap.org/>.
- Martinez S, Garcia-Alfaro J, Cuppens F, Cuppens-Boulahia N, Cabot J. A model-driven approach for the extraction of network access-control policies. In: Satellite events at the MoDELS 2012 conference (MDSEC 2012); 2012.
- Mayer A, Wool A, Ziskind E. Fang: a firewall analysis engine. In: IEEE symposium on security and privacy; 2000. p. 177–87.
- Meng X, Jiang G, Zhang H, Chen H, Yoshihira K. Automatic profiling of network event sequences: algorithm and applications. In: 27th conference on computer communications (INFOCOM 2008); 2008. p. 1–9.
- Neira Ayuso P. Netfilter's connection tracking system: login. The USENIX Magazine 2006;31(3):40–5.
- Preda S, Cuppens-Boulahia N, Cuppens F, Garcia-Alfaro J, Toutain L. Model-driven security policy deployment: property oriented approach. In: Second international symposium on engineering secure software and systems (ESSoS 2010); 2010. p. 123–39.
- Preda S, Cuppens-Boulahia N, Cuppens F, Garcia-Alfaro J, Toutain L. Dynamic deployment of context-aware access control policies for constrained security devices. Journal of Systems and Software 2011;84(7):1144–59.
- HPING3. Security tool. URL: <http://www.hping.org/hping3.html>.
- Srinivasan V, Suri S, Varghese G. Packet classification using tuple space search. Computer Communication Review 1999;29(4): 135–46.
- The NetFilter Project: Firewalling, NAT and Packet Mangling for Linux. URL: <http://www.netfilter.org/>.
- Treurniet J. Detecting lowprofile scans in TCP anomaly event data. In: Fourth annual conference on privacy, security and trust (PST 2006); 2006. p. 1–8.
- Venkatamohan B. Automated implementation of stateful firewalls in Linux. Master's thesis, North Carolina State University; 2011.
- Xtext. Language development made easy. URL: <http://www.eclipse.org/Xtext/>.
- Youssef N, Bouhoula A. Dealing with stateful firewall checking. In: Digital information and communication technology and its applications; 2011. p. 493–507.
- Yuan L, Mai J, Su Z, Chen H, Chuah C, Mohapatra P. FIREMAN: a toolkit for firewall modeling and analysis. In: IEEE symposium on security and privacy; 2006. p. 199–213.
- Joaquin Garcia-Alfaro** is associate professor at TELECOM SudParis, Networks and Telecommunication Services department. He holds an Engineering degree in Computer Science and a PhD. His research includes management of security policies, analysis of vulnerabilities and enforcement of countermeasures.
- Frédéric Cuppens** is a full professor at TELECOM Bretagne, LUSSI department. He holds an Engineering degree in Computer Science, a PhD and an HDR. His research includes formal models, security policies, access control and intrusion detection.
- Nora Cuppens-Boulahia** is an associate researcher at TELECOM Bretagne, LUSSI department. She holds an Engineering degree in Computer Science, a PhD and an HDR. Her research includes formalisation of security properties, cryptographic protocol analysis, formal validation and threat assessment.
- Salvador Martinez** is a PhD candidate at the AtlanMod team of the École des Mines de Nantes. His research interests include model-driven security, modeldriven reverse engineering (particularly of security-related aspects) and model transformation languages.
- Jordi Cabot** is currently leading the AtlanMod team, an INRIA research group at École des Mines de Nantes. He holds an Engineering degree in Computer Science and a PhD. His research interests include conceptual modeling, model-driven and web engineering, and formal verification.