A Formal Composition of a Distributed System with its Security Policy

Wadie Krombi, Mohamed Mustapha Kabbaj and Mohammed Erradi

ENSIAS, Mohammed V-Souissi University
Rabat, Morocco

Abstract

Nowadays, information systems are becoming a vital and strategic component of any organization. However, in most cases, these systems are designed and implemented without taking into consideration security aspects. To ensure a certain level of security, the behavior of a system must be controlled by a "security policy". The objective of this work is: Given a system S and a security policy P how can we generate a system Sp which is a secure version of S? Based on the fact that a security policy is a set of rules, we propose an approach to build an automaton modeling a security policy. Then we propose an approach for modeling a system with the same formalism. Finally, we suggest a composition model of a system with a security policy. The suggested approach is illustrated using a firewall security policy and a distributed system consisting of network elements (servers, workstations …).

Keywords: *Security Policy, System, Automata, Composition, Firewall, Security Rule.*

1. Introduction

Information systems are the nerve center of any modern organization. However, in most cases, these systems are designed and implemented without taking into consideration security aspects. This makes them vulnerable to attacks and intrusions that may affect their normal functioning. So, we can easily recognize the importance of providing these systems the appropriate level of protection by establishing security policies. [1]

To ensure a certain level of security, the behavior of a system must be controlled by a "security policy." The security policy of a system specifies the set of laws, rules and practices that regulate how sensitive information and other resources are managed, protected and distributed within a specific system. It shall identify the security objectives of the system and the threats to the system. [2]

The ultimate objective of our work, given a system S and a security policy P, is to generate a system Sp which is a secure version of S (Fig.1). That's why we need to develop a formal and systematic approach to compose a system S with a security policy P. Therefore, this composition must ensure compliance of the system obtained from this composition with the initial system and the security policy.

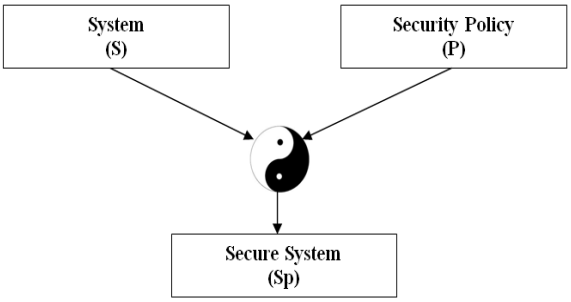

Fig.1 Composition of a system and a security policy

To do this, we propose to use the same formal model to express both the system S and the security policy P. This formalization will allow abstracting of the security policy, managing its complexity, detecting and resolving conflicts and ensuring that all security objectives are covered by the measures previously identified. [3]

To solve this problem, we propose an approach based on the following main steps:

- Modeling a security policy by an automaton
- Modeling a distributed system by a global automaton
- Compose S and P to get Sp a secure system as a result of applying the security policy to the initial system.

The rest of this paper is organized as follows: Section 2 presents a state of the art on security policies modeling. Section 3 begins by recalling some basics on firewalls and automata and presents afterwards the security policy transformation process to an automaton. Section 4 is dedicated to the modeling of a system by an automaton. In section 5 we propose a composition model of a system with its security policy using the same "automata" formalism. Finally, we conclude and propose some future work.

2. State of the art

Several research studies that address the network security policies focus on firewalls. In [4], the authors propose the "Diverse Firewall Design" method that has as its objective the discovery of all functional discrepancies between different implementations of a firewall security policy. The fundamental data structure which is used to model this policy is called "Firewall Decision Diagram" (FDD) [5]. FDD maps each packet to a decision by testing the packet throughout the diagram from the root to a terminal node. This indicates the decision to take by the firewall for the current packet. Each non-terminal node in a FDD specifies a test on a field in the packet, and each branch descending from that node corresponds to the possible values of this field.

In [6][7], the authors present a set of techniques and algorithms that allow automatic discovery of anomalies in the firewall security policy. The firewall security policy is represented by a graph that is a tree with a single root called "Policy tree" in [6] and "Decision tree" in [7], where each node represents a field of the filtering rule; each branch is a possible value of the field. Each path in the tree begins with the root and ends with a leaf; it represents a filtering rule in the security policy and vice versa. Rules having the same field value related to a specific node share the same branch that represents this value. The leaves are actions that can be executed (*Accept*, *Deny*).

In [8], the authors introduce Fireman, a "toolkit" of static analysis for modeling and analysis of firewalls. By treating firewall configurations as specialized programs, Fireman applies a set of static analysis techniques to examine the configuration errors such as policy violations and inconsistencies both within an individual or distributed firewall. Fireman is implemented by modeling firewall rules using “Binary Decision Diagrams” (BDD) [9] to represent predicates and perform all the set of the available operations.

In these studies no distinction was made between the system (to secure) and the security policy to be applied. Also, these proposed approaches do not address the formal specification of the system from the functional point of view.

In [10], the authors propose a framework that makes it possible to automatically generating test sequences to validate the conformance of a security policy. In this framework the behavior of the system, without taking into account aspects related to security, is separately specified as an extended finite state machine (Extended Finite States Machine - EFSM) [11] while the security policy is specified based on another formalism which is the OrBAC model [12].

3. Modeling a security policy

3.1 Firewalls and automata

Firewall is one of the critical and important security components of information systems; it is a system that protects resources of a private network against intrusions or threats that may come from other networks (eg Internet). A firewall is designed to logically separate networks with different levels and different security requirements. The separation is usually done on the basis of rules governing permitted communications between networks.

The behavior of a firewall is controlled by its security policy which is represented by an ordered list of security rules that defines the actions to execute each time a packet passes through the firewall. A packet is defined as a tuple of a finite number of network fields such as source IP address, destination IP address, port number, protocol, etc.

A firewall security rule (also called filtering rule) is expressed in the form: if certain conditions are met, an action must be executed to allow access or to refuse it. A rule can be represented as *"Condition => Action":*

- The *"Condition"* field is a boolean expression applied to the various fields of the packet. It consists of a set of filtering fields. These are the possible values of the corresponding fields in the packets of the current network traffic that matches this rule.
- The *"Action"* field can be *"Accept"* which allows the packet through the firewall or *"Deny"* which blocks the packet.

The crossing of a packet is allowed or blocked by a specific rule if the header information’s of the packet match to all rule fields. Otherwise, the next rule is examined and the process is repeated until a security rule that matches is found. If no rule matches the packet that passes through the firewall, a default policy is applied.

A finite state automaton (more briefly, automaton) can be formally defined by $A=(\Sigma, Q, q_0, Q_f, \delta)$, where Σ is a finite set of events (also called alphabet), Q is a finite set of states, q_0 is the initial state and Q_f is the set of final states, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function, where $\delta(q, \sigma)=r$ means that the execution of the event σ from state q leads to state r [13]. We use the following two notations:

- For a sequence of events $\lambda=\sigma1,\ldots,\sigma p$, $\delta(q,\lambda)=r$ means that if q is the current state, then the consecutive execution of $\sigma1,\ldots,\sigma p$ leads to state r;
- For a set of events $S=\{\sigma1, \ldots, \sigma p\}$, $\delta(q,S)=r$ means that if q is the current state, then every event of S leads to state r.

An automaton A can be represented by a graph whose nodes and arcs represent the states and the transitions of A, respectively. An arc from node q to node r labeled by the event σ represents the transition $\delta(q, \sigma)=r$.

Several arcs labeled $\sigma_1,\ldots,\sigma_n$ linking the same pair of states *(q,r)* can be represented by a single arc labeled by the set $\{\sigma_1,\ldots,\sigma_n\}$. A finite event sequence (more briefly, sequence) is accepted by *A* if it starts in the initial state q_0 and terminates in any state of *A*. The language of *A*, denoted L_A, is the set of sequences accepted by *A*.

3.2 Basic principle of the modeling approach

As mentioned above, when a firewall receives a packet, it compares the value of each field in the packet header with the one corresponding to the same field in the security policy. Thus, the firewall compares the information in the packet header fields and those filtering the current rule. If there is a match then the action of this rule is applied to this packet. Otherwise, the firewall examines the packet by the following rule and the process is repeated until a security rule that matches the packet will be found.

Let Σ the alphabet consisting of the digits *[0…9]* and the symbol *"."*. *L(IP)* the IP addresses language whose words are of the form *"a.b.c.d"* with $a,b,c,d \in [0,255]$. *L(Port)* the port language whose words are numbers between *0* and *62535*. *L(Protocol)* the protocols language whose words representing a protocol (TCP, UDP…).

Let *L(Packet)* the language defined as the concatenation of *L(IP) L(IP) L(Port) L(Protocol).* A packet whose header is composed of a source IP address, destination IP address, port number and protocol is a word of *L(Packet).*

Let *L(Packet-Ri)* the language defined as the concatenation *L(IPsrc-Ri) L(IPdst-Ri) L(Port-Ri) L(Protocol-Ri),* with:

- *L(IPsrc-Ri)* the subset of words in *L(IP)* consisting of IP addresses that match the filtering field condition of the source IP address field in rule *Ri.*
- *L(IPdst-Ri)* the subset of words in *L(IP)* consisting of IP addresses that match the filtering field condition of the destination IP address field in *Ri.*
- *L(Port-Ri)* the subset of words in *L(Port)* consisting of port numbers that match the filtering field condition of the port number field in *Ri.*
- *L(Protocol-Ri)* the subset of words in *L(Protocol)* consisting of protocols that match the filtering field condition of the protocol field in *Ri.*

A packet whose header is respectively composed of a source IP address, destination IP address, port number and protocol that correspond to filtering fields of a rule *Ri* is a word belonging to language *L(Packet-Ri).* The condition of a filtering field may be either a single value or a range of values or "Any" which indicates any value.

The basic idea behind modeling a security policy by an automaton is as follows: in a given state corresponding to a network filtering field (IPsrc, IPdst, Port or Protocol), the automaton read a value of a network field. The label of this transition is in fact the condition which permits the transition from one state to another.

A security rule *Ri* of a firewall can be modeled by the automaton shown in Fig.2. This automaton recognizes words (packets) of *L(Packet-Ri)* and after "consuming" them, it ends up in a final state that indicates the action to execute for this packet. The other words (packets) of *L(Packet)* not belonging to the language *L(Packet-Ri)* should be examined by the following rule *Ri+1*.

We call "*positive transition*" a transition labeled by the filtering condition related to a filtering field of a rule. We call "*negative transition*" a transition labeled by the complement of filtering condition related to a filtering field of a rule. The label of a negative transition will be denoted by *"! Condition*" We call "*positive path*" of rule *Ri* the only path from initial state of the rule *Ri* to its final state that indicates the action to execute if a packet matches rule *Ri*. This path is a sequence of positive transitions. We call "*negative path*" of rule *Ri* a path from initial state of rule *Ri* to initial state of rule *Ri+1*. This path has a single negative transition whose label is a non verified condition by one of the filtering fields of rule *Ri*. A negative path may also include special states and transitions which we call "*consumption states and transitions*". The condition of a consumption transition is always verified and is labeled "*Any*". The usefulness of consumption states and transitions will be described later.

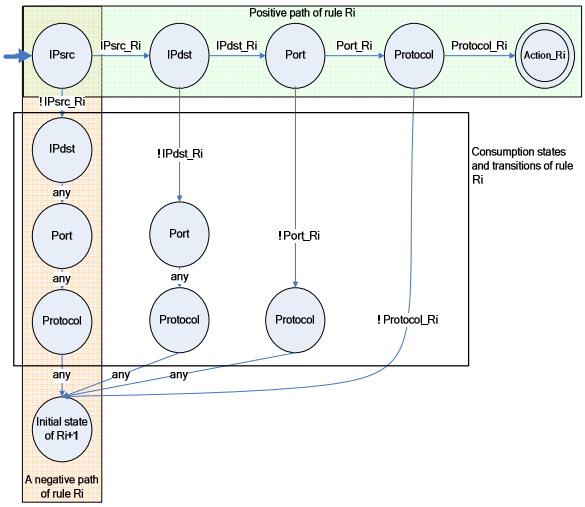

Fig. 2 Automaton modeling Rule *Ri*

Suppose that the firewall is about to inspect a given packet belonging to *L(Packet)* whose header is composed of values *[IPsrc-Packet,IPdst-Packet,Port-Packet,Protocol-Packet]*. Suppose also that the firewall is at the stage of analyzing this packet by rule *Ri*. If we suppose, for example, that *IPsrc-Packet* does not belong to *L(IPsrc-Ri)* then: whatever *IPdst-Packet, Port-Packet* and *Protocol-Packet* values, the current packet does not match rule *Ri*. Thus, according to the firewall filtering process, the packet

must then be examined by the following rule *Ri+1*. Passing from rule *Ri* to rule *Ri+1* can be modeled by a negative path consisting of the following transitions and states:

- A negative transition from the *IPsrc* state of rule *Ri* to a new consumption state *IPdst*. This transition is labeled *"! IPsrc-Ri"*. This label describe the non-membership of *IPsrc-Packet* to *L(IPsrc-Ri)*.
- A consumption transition from the last created consumption state *IPdst* to a new consumption state *Port*.
- A consumption transition from the last created state *Port* to a new consumption state *Protocol*.
- A consumption transition from the last created state *Protocol* to first state of the following rule *Ri+1*.

Consumption states and transitions are created when a packet does not match a rule *Ri* and must be examined by the following rule *Ri+1*. Their role is to ensure that a negative path that connects the rule *Ri* to the rule *Ri+1* passes through "exactly" the same states sequence as the positive path of the rule *Ri* and in the same order. Thus, after inspecting the state related to the last filtering field of the packet, the automaton is in one of the following states:

- In the final state *Action-Ri* if the packet matches *Ri*.
- In the initial state of rule *Ri+1* if one of the fields of the packet don't match a filtering condition of *Ri*.

3.3 Construction process of an automaton from a security policy

In this section we will describe the construction process of an automaton from a security policy. We assume that the policy has, necessarily, a "default rule" (the last one).

3.3.1 Policy with one rule

In this case, the policy is only constituted of the default rule (Table 1). Fig.3 presents the automaton corresponding to the policy of Table 1.

Table 1: Policy with one rule

IPsrc	IPdst	Port	Protocol	Action
Any	Any	Any	Any	Action-R1

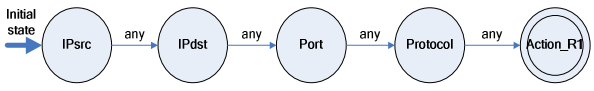

Fig. 3 Automaton modeling Table 1 policy

This policy can be interpreted as follows: for a packet received by firewall, whatever *IPsrc*, *IPdst*, *Port* and *Protocol* values, the action to execute is *"Action-R1"*. Thus, upon receiving a packet by firewall, it has no need to check any value of rule filtering fields and the automaton goes directly to final state "*Action-R1*". Therefore, the default rule can be modeled by the reduced automaton shown in Fig.4.

Fig. 4 Reduced automaton modeling Table 1 policy

3.3.2 Policy with two rules

In the case of a policy with two rules (Table 2), according to the process described earlier to model a security policy and on modeling default rule, the corresponding automaton of this policy is shown in Fig.5.

Table 2: Policy with two rules

IPsrc	IPdst	Port	Protocol	Action
IPsrc-R1	IPdst-R1	Port-R1	Protocol-R1	Action-R1
Any	Any	Any	Any	Action-R2

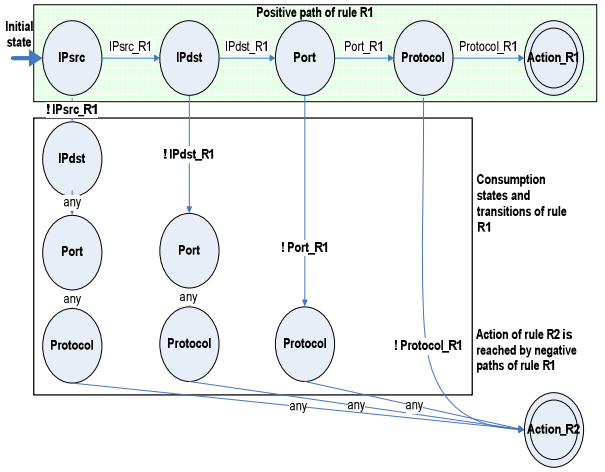

Fig. 5 Automaton modeling Table 2 policy

Recall that consumption states and transitions are created when a packet does not match a rule *Ri* and must be examined by the following rule *Ri+1*. But in this case the following rule *R2* is the default one. So when a field in the packet does not match a filtering condition of the current rule *R1* the automaton changes state via a negative transition to the final state *Action-R2*. Security policy of Table 2 can thus be modeled by the reduced automaton represented in Fig.6.

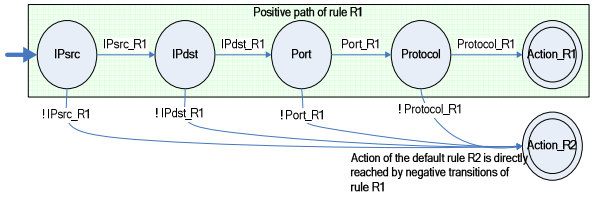

Fig. 6 Reduced automaton modeling Table 2 policy

3.3.3 Policy with more than two rules

In the case of a policy with *m* rules and *m>2*, the automaton is obtained using the following process:

- Creating and concatenating of partial automata of each rule with the following one until rule *Rm-1* (using the process described in Section 3.2 and illustrated in Fig.2)
- Creating positive path of rule *Rm-1*
- Creating negative transitions of rule *Rm-1* that lead to the default action of the last rule *Rm* (using the process described in Section 3.3.1 and illustrated in Fig.4).

Table 3 is an example of a policy with rules 3 and Figure 7 shows the equivalent automaton.

Table 3: Policy with tree rules

IPsrc	IPdst	Port	Protocol	Action
IPsrc-R1	IPdst-R1	Port-R1	Protocol-R1	Action-R1
IPsrc-R2	IPdst-R2	Port-R2	Protocol-R2	Action-R2
Any	Any	Any	Any	Action-R3

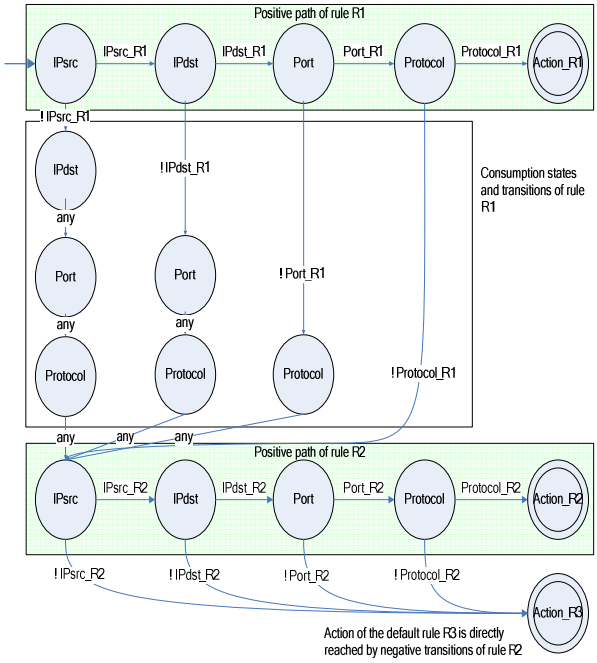

Fig. 7 Automaton modeling Table 3 policy

3.4 Case study

Consider a company network which is connected to the internet and we want to protect it with a firewall (Fig.8). The internal network is composed of two segments: The LAN users (192.168.10.0/24) and the DMZ hosting the Web server (212.217.65.201) and FTP server (212.217.65.202). The company has a branch office network (194.204.201.0/28) connected to company headquarters internal network through internet.

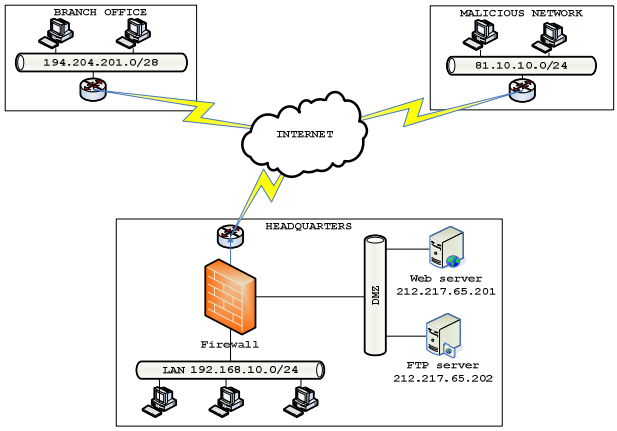

Fig. 8 Example of company network secured by a firewall

This company security policy requirements are as follows:

- Access to the Web server is allowed to all.
- FTP access is allowed only from company's internal LANs (headquarters and branch office LANs).
- Users on the internal LAN of the company's headquarters are allowed to access to the entire internet network with the exception of the malicious network 81.10.10.0/24.

Table 4 represents firewall security rules corresponding to the above security requirements of the company.

Table 4: Firewall security policy of the company

IPsrc	IPdst	Port	Protocol	Action
Any	212.217.65.201	80	TCP	Accept
192.168.10.0/24	81.10.10.0/24	Any	Any	Deny
194.204.201.0/28	212.217.65.202	21	Any	Accept
192.168.10.0/24	Any	Any	Any	Accept
Any	Any	Any	Any	Deny

By applying automaton construction process to Table 4 security policy, we obtain the equivalent automaton of this policy (Fig. 9). Thus, this automaton is able to recognize what action to execute for a packet by performing: at least 3 transitions if the packet matches the rule *R1* and maximum 9 transitions if the packet matches the last default rule *R5*. Each transition corresponds in fact to a test done over a filtering condition of a security rule. For the same studied example, a packet filtered by the conventional process requires firewall to perform at least 4 tests if the packet matches the rule *R1* and up to 25 tests if the packet matches the last default rule *R5*. We deduce that by using our automaton model, the load generated by the filtering process of the firewall studied in this example can be reduced by *25%* to *64%*.

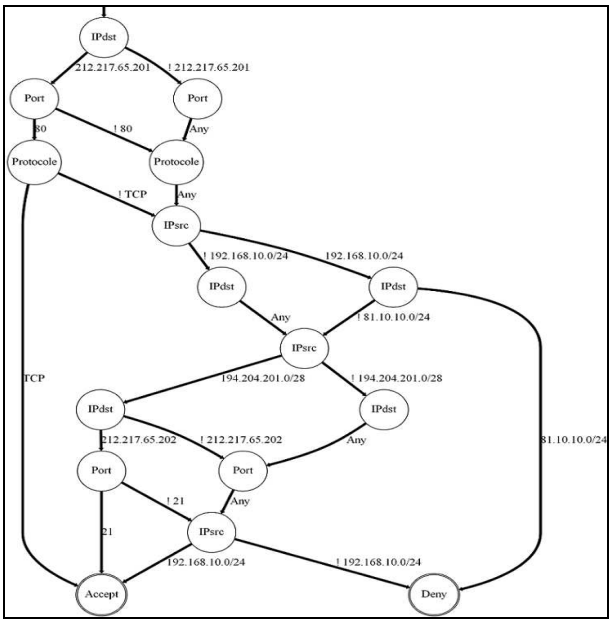

Fig. 9 Automaton modeling Table 4 policy

4. A client-server system modeling

In this paper, a system is considered as a set of network components which may communicate with each other. We also limit the scope of this paper to TCP/IP communications according to Client-Server model.
The objective in this section is to be able to model a system of N components and its overall behavior in order to know the exact state of each component at a given time.
To do this, we will begin by studying a basic system which consists of two components one as a client and the other as a server. Then we will extend our study in order to be able to generalize the modeling of a system of N components which can be at the same time in Client mode and/or in server mode, representing the real behavior of a system.

4.1 Modeling communication of two components

4.1.1 Construction process

In order to model the process of establishing a client-server communication between two components of a system, we have to model:

- The client process of a system component
- The server process of a system component
- The establishing communication process between the client and the server components

We can model the *Client* and *Server* process of a system component by automata describing their different possible states and transitions that can trigger a change of state of such a component.

According to state diagram of a TCP connection [14]:

- As a client, a system component can be in one of the following states:
  - *« CLOSED »:* is a fictional state, it represents the state when there is no connection.
  - *« SYN-SENT »* (or *« SYN-CLIENT »* for client synchronization): represents waiting for a matching connection request after having sent a connection request (*SYN*).
  - *« ESTABLISHED »* (or *« CLIENT »*): represents an open connection as a client with a server.
- As a server, a system component can be in one of the following states:
  - *« CLOSED »*
  - *« LISTEN »* : represents waiting for a connection request from a client
  - *« SYN-RECEIVED »* (or *« SYN-SERVER »* for server synchronization): represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.
  - *« ESTABLISHED »* (or *« SERVER »*): represents an open connection as a server with a client.

We can define transitions that trigger state changes of a system component based on those of the state diagram of a TCP connection.
Figures 10 and 11 show, respectively, automata modeling the behavior of a component of a system: the first one as a client and the second one as a server.

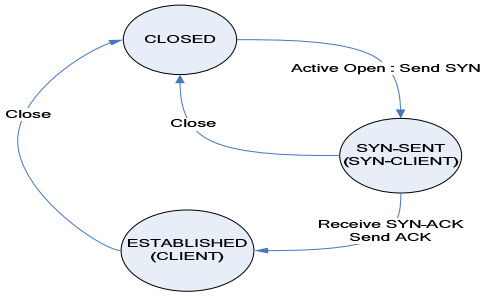

Fig. 10 A system component automaton in *Client* mode

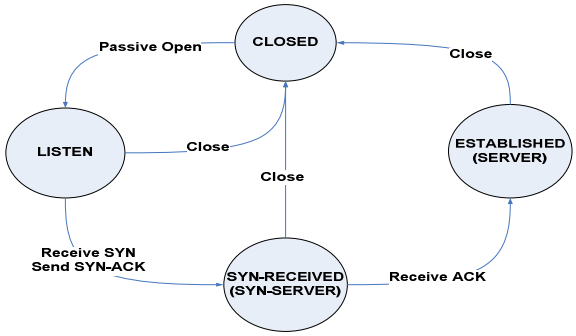

Fig. 11 A system component automaton in *Server* mode

To obtain the automaton modeling the communication between two components one as a client and the other as a server it is necessary to compose both automata modeling the behavior of each one of these components separately. To achieve this goal, first we will use *Promela* language for formal description of client and server processes modeled by the automata of Figures.10 and 11, then using the *Spin* tool we will get necessary sequencing events for establishing communication between client and server afterwards we will deduct from it the automaton modeling the overall behavior of a communication between a client and a server.

Figures 12 and 13 show, respectively, the *Promela* formal description of client (server) process of a system component. Note that the status of the client (server) component is designated by the *"StateC" ("StateS")* variable. Recall that in *Promela*, the symbols *"!"* And *"?"* mean, respectively, sending and receiving messages using the specified parameter set before the channel symbol.

```
Proctype C()
{
Byte message;
Byte stateC=Closed;

do::
   if
    ::(stateC == Closed) =>
         atomic(
         C2S!Syn;
         stateC=Syn_Client;
         )
    ::(stateC == Syn_Client) =>
         atomic(
         S2C?message;
         if::(message == SynAck) =>
            C2S!Ack;
            staceC = Client;
         fi;
         )
   fi;
od;
}
```

Fig. 12 Formal description of the C*lient* process in *Promela*

```
Proctype S()
{
Byte message;
Byte stateS=Closed;

do::
   if
    ::(stateS == Closed) =>
         atomic(
         if::Passive_open =>
            staceS = Listening;
         fi;
         )
    ::(stateS == Syn_Listening) =>
         atomic(
         if::C2S?message;
         if::(message == Syn) =>
         S2C!SynAck;
         StateS = Syn_Server;
         fi;
         fi;
         )
    ::(stateS == Syn_Server) =>
         atomic(
         C2S?message;
         if::(message  ==  Ack)  =>
         StateS = Server;
         fi;
         )

   fi;
od;
}
```

Fig. 13 Formal description of the S*erver* process in *Promela*

After initialization of both client and server processes by *Promela* by the execution of *run(C)* and *run(S)* commands, we can obtain the communication process between client and server using *Spin*. Figure 14 shows the output file describing the sequencing of the communication process between client and server.

```
 1:    proc  0 (:init:) tcp.pml:92 (state 1)    [(run C())]
 2:    proc  1 (C) tcp.pml:13 (state 1)    [((stateC==Closed))]
 3:    proc  0 (:init:) tcp.pml:93 (state 2)    [(run S())]
 4:    proc  2 (S) tcp.pml:43 (state 1)    [((stateS==Closed))]
 5:    proc  2 (S) tcp.pml:48 (state 2)    [(Passive_open)]
 6:    proc  2 (S) tcp.pml:49 (state 3)    [stateS = Listening]
 7:    proc  2 (S) tcp.pml:53 (state 7)    [((stateS==Listening))]
 8:    proc  1 (C) tcp.pml:16 (state 2)    [C2S!Syn]
 9:    proc  1 (C) tcp.pml:17 (state 3)    [stateC = Syn_Client]
10:    proc  1 (C) tcp.pml:19 (state 5)    [((stateC==Syn_Client))]
11:    proc  2 (S) tcp.pml:58 (state 8)    [C2S?message]
12:    proc  2 (S) tcp.pml:59 (state 9)    [((message==Syn))]
13:    proc  2 (S) tcp.pml:60 (state 10)   [S2C!SynAck]
14:    proc  2 (S) tcp.pml:61 (state 11)   [stateS = Syn_Serveur]
15:    proc  2 (S) tcp.pml:65 (state 17)   [(((stateS==Syn_Serveur)))]
16:    proc  1 (C) tcp.pml:21 (state 6)    [S2C?message]
17:    proc  1 (C) tcp.pml:23 (state 7)    [((message==SynAck))]
18:    proc  1 (C) tcp.pml:24 (state 8)    [C2S!Ack]
19:    proc  1 (C) tcp.pml:25 (state 9)    [stateC = Client]
20:    proc  2 (S) tcp.pml:67 (state 18)   [C2S?message]
21:    proc  2 (S) tcp.pml:68 (state 19)   [((message==Ack))]
22:    proc  2 (S) tcp.pml:70 (state 20)   [stateS = Serveur]
23:    proc  2 (S) tcp.pml:40 (state 26)
24:    proc  1 (C) tcp.pml:11 (state 15)
25:    proc  0 (:init:) tcp.pml:95 (state 3)
26:
```

Fig. 14 Output file describing the *client-server* communication process using *Spin*

From this sequence we can now obtain the automaton modeling the overall behavior of a communication between a client and a server (Fig. 15).

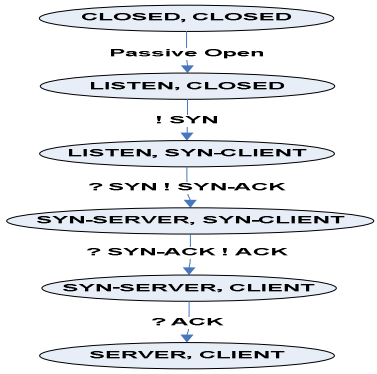

Fig. 15 Automaton modeling a *client-server* communication

4.1.2 Automaton interpretation

A state of the automaton is represented by a *couple* which gives indication of the state of the server component and the client component during communication.

In the first two states of this automaton, at least one of the two components of the system is inactive (CLOSED). In both states, no communication can be initiated between these two components.

The remaining other states of the automaton represent each one a *couple* indicating a communication step between two "active" components of the system. So we can describe the following couples: (Table 5). Note that when an “active” component of the system is not part of a communication, it is necessarily in the "*LISTEN*" state.

Table 5: Couples types representing a state in Fig. 17 automaton

Name	Server state	Client state
Couple1	LISTEN	SYN-CLIENT
Couple2	SYN-SERVER	SYN-CLIENT
Couple3	SYN-SERVER	CLIENT
Couple4	SERVER	CLIENT

4.2 Automaton modeling a system of two components

Since the system is a set of several components that can communicate with each other and that each communication is identified by two components of the system, then a given state in the automaton that will model the overall behavior of the system will represent all possible states of communication between its various components.
Thus, a given state of the system must be able to indicate:

- The list of all components of the system that are in a "*LISTEN*" state, which we call "*Listeners*"
- The list of all type 1 system components couples in which the server is in a "*LISTEN*" state and the client is in a "*SYN-CLIENT*" state, which we call "*List-1*".
- The list of all type 2 system components couples in which the server is in a "*SYN-SERVER*" state and the client is in a "*SYN-CLIENT*" state, which we call "*List-2*".
- The list of all type 3 system components couples in which the server is in a "*SYN-SERVER*" state and the client is in a "*CLIENT*" state which we call "*List-3*".
- The list of all type 4 system components couples in which the server is in a "*SERVER*" state and the client is in a "*CLIENT*" state which we call "*List-4*".

We can deduce that a given state of the system is completely identified by the knowledge of the elements that comprise the five lists previously defined.
Let us introduce *Tsys* the 5-tuple consisting of these lists, we have: *Tsys = (Listeners, List-1, List-2, List-3, List-4).*
In a state of communication that represents the server *S* in a "*LISTEN*" state, the current server *S* is added to the "*Listeners*" list and the system state is identified by the 5-tuple of lists *Tsys* and we have:

- *Tsys .Listeners= Tsys .Listeners.ADD(S)= S*
- *Tsys .List-1=null*
- *Tsys .List-2=null*
- *Tsys .List-3=null*
- *Tsys .List-4=null*

Note *Update-Sys-0* the function which allows this first state change in the system (ie in the 5-tuple *Tsys*). According to the automaton, this function is executed and provides a state change to the system whenever a transition "*Passive open*" is triggered.
In a state of communication that represents a type 1 couple *(S,C),* this one is added to *« List-1 »* and *« Listeners »* contains always the server *S* and the system state is identified by the new values of *Tsys* and we have:

- *Tsys .Listeners= S*
- *Tsys .List-1= Tsys .List-1.ADD(S,C)=(S,C)*
- *Tsys .List-2=null*
- *Tsys .List-3=null*
- *Tsys .List-4=null*

Note *Update-Sys-1* the function which allows this second state change in the system. According to the automaton, this function is executed and provides a state change to the system whenever a transition "*SYN*" is triggered.
For every i=2,..,4: in a state of communication that represents a type *i* couple *(S,C),* this one is added to *List-i* and the type *'i-1'* couple *(S,C)* is deleted from *List-'i-1',* this is necessary to represent the state change made in a communication between two components of the system and we particularly have:

- *Tsys .List-'i-1'= Tsys .List-'i-1'.DELETE(S,C)*
- *Tsys .List-i= Tsys .List-i.ADD(S,C)=(S,C)*

Note *Update-Sys-2 (*respectively *Update-Sys-3, Update-Sys-4)* the function which is executed and provides a state change to system whenever a transition *"?SYN!SYN-ACK"* (respectively *"?SYN-ACK!ACK", "?ACK")* is triggered.
In the particular case of a system consisting of two components, system automaton can be obtained from the one modeling communication between two components by carrying out a special "renaming" operation of its states by the use of the five functions we just define and which manipulate the global state of the system which is completely identified by the knowledge of the *Tsys* elements. Apart from the initial state that represents the system in a state of inactivity, any state in the new system automaton is now labeled with a function that is executed after a given transition and which change the overall state of the system by changing *Tsys* elements. (Fig. 16)

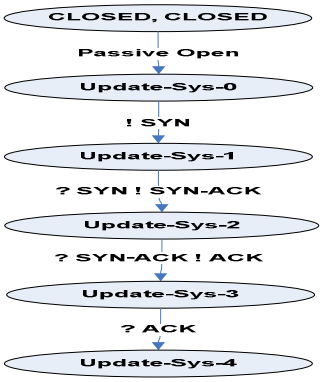

Fig. 16 Automaton modeling a system of two components

4.3 Automaton modeling a system of N components

To obtain the automaton modeling a system comprised of N components, we just need to take as basis the one modeling a system of two components and supplement it by new transitions which can indicate that at any time and in any state of the system: any component may initiate several simultaneous communication with other components as client, or that a component can be a server for multiple clients at once.

In order to better justify this, consider the example of a state of communication between two components of the system. Suppose that we are in the *Update-SYS-1* state and another server changes state from the "*CLOSED*" state to the "*LISTEN*" one by the "*Passive open*" transition, then we can notice that if we want this situation to be represented in the automaton we want to build, it is necessary that the new automaton contains a transition "*Passive open*" from the "*Update-Sys-1*" state to the "*Update-Sys-0*" state. This transition does not exist in the automaton of two components (Fig. 16).

So, to be able to model all possible cases of communications between several components of the system, the global automaton modeling the system must be supplemented by transitions that reflect the change of state of the system to any other one. The construction of this automaton is done according to the following algorithm.

```
Input: Old_Aut (Initial automaton of two components)
Output: New_Aut (Resulting automaton of N components)
BEGIN
New_Aut=Old_aut
Old_States=List of all states of Old_Aut
Old_Transtions=List of all transitions of Old_Aut
FOR every transition T of Old_Transitions DO
        FOR every state S of Old_States DO
                IF (S ≠ T.start) DO
                        New_T=new Transition
                        New_T.start=S
                        New_T.end=T.end
                        New_T.Label=T.Label
                        New_Aut.ADD_Transition(New_T)
                END-IF
        END-FOR
END-FOR
END
```

By applying this algorithm to Fig. 16 automaton we obtain the one modeling a system of N components. (Fig.17)

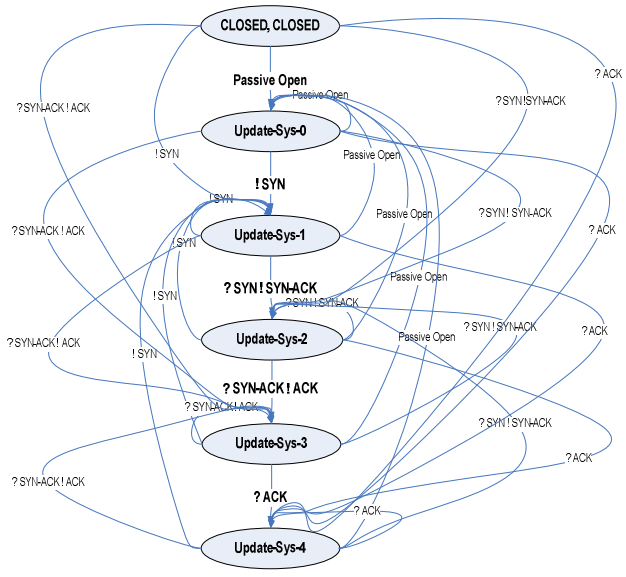

Fig. 17 Automaton modeling a system of N components

5. Composing a system and a security policy

At this stage we are able to model by the same formalism of automata: a security policy and the behavior of a system. We propose now to model the composition of a system with a security policy, more precisely we propose to model a given system controlled by a given security policy by the same formalism of automata.

In the following, for the sake of simplicity and without loss of generality, we assume that only the TCP protocol is used. TCP is "connection-oriented protocol". This means that when a packet is sent, it has information indicating it is the first packet of a given communication or if it is a suite to a previously received packet.

In [15] an example is given to explain this aspect of TCP connections. In Figure 18, the host whose IP address is *192.168.1.1* initiated communication with the one whose IP address is *1.2.3.4*. The corresponding packet then contains "*SYN*" flag. Packets following this first exchange will all contain "*ACK*" flag. In the following, we will not specify connection flags (*SYN*, *ACK*) and will consider that, in the case of firewall filtering rules, only authorizations related to connections initialization are specified. The corresponding "replies" packets are implicitly accepted, thereby filtering rules are supposed being applied to packets with the *SYN* flag and is expected to present a default rule for packets with the *ACK* flag.

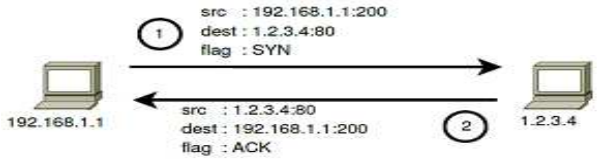

Fig. 18 Example of a TCP request for connection establishment

From this last assumption, we can deduce that in order to obtain the automaton modeling the composition *Sp* of a system *S* and a security policy *P*, we need simply to express the fact that the security policy is controlling "*SYN*" transitions of the automaton *S*. Thus, any connection request (*SYN* Send) initiated by a client component of the system addressed to a server component of this same system will not be directly addressed to the destination component (the server), it will first be relayed to the automaton of the security policy for analysis: if at the end of this analysis the connection request is authorized by the security policy *P* then the connection establishment process continues normally in the automaton of the system *S*. Otherwise, the connection establishment process is interrupted and can't be continued in the system.

The automaton *Sp* modeling the system *S* secured by the security policy *P* is obtained by applying the following algorithm to *S* and *P*. Figure 19 illustrates the result of applying the algorithm to a *SYN* transition.

```
Input: System S, Security policy P
Output: Secure system Sp
BEGIN
Sp=S
FOR every transition T of S DO
        IF (T.label=SYN) DO
                New_T1=new Transition
                New_T1.start=T.start
                New_T1.end=P.Initial_State
                New_T1.Label=SYN
                Sp.ADD_Transition(New_T1)
                New_T2=new Transition
                New_T2.start=P.Final_Accept_State
                New_T2.end=T.end
                New_T2.Label=SYN
                Sp.ADD_Transition(New_T2)
                Sp.DELETE_Transition(T)
        END-IF
END-FOR
END
```

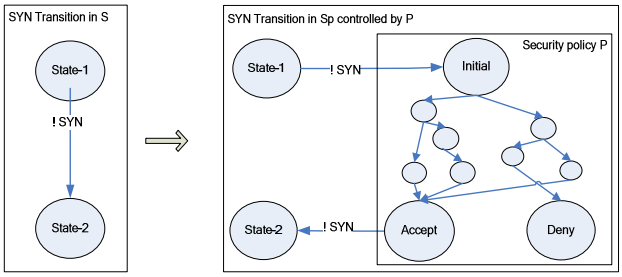

Fig. 19 Result of applying the algorithm to a *SYN* transition

6. Conclusion and future work

The overall objective of our work is to model a system *S* and its security policy *P* by the same formalism of automata and then compose both of them in order to generate a new System *Sp* which is a secure version of *S* in conformance to *P*. Such a separation between the system specification and the security policy requirements allows in one hand the improvement of systems scalability and in the other hand the reuse of security policies.

In this work we showed how to express a security policy as an automaton. We then proposed a model of a system comprised of several network components using the same automata formalism having as a basis the TCP state diagram. Finally, we showed how it is possible to compose a system and a security policy in order to obtain as a result a secured version of this system controlled by this security policy still using the same automata formalism.

By combining the results obtained in this work and a judicious use of the theory of automata richness and rigor, we consider as future work to study other aspects related to security of systems, such as: how to model communication between components belonging to different systems which are separately controlled by different and independent security policies. Once the modeling assumed to be realized, it would be interesting to study how to ensure that such communication is in conformance with the various security policies that control these different systems.

References

[1] N. Dausque, “PSSI & CAPSEC : Politique de Sécurité des Systèmes d’Information & Comment Adapter une Politique de Sécurité pour les Entités du CNRS,” CNRS/UREC for CAPSEC group, Mars 2005

[2] “Information Technology Security Evaluation Criteria (ITSEC)”, Office for Official Publications of the European Communities, Luxembourg, p.20, v1.2, June 1991

[3] A. Baina, “Contrôle d'Accès pour les Grandes Infrastructures Critiques” PhD thesis, University of Toulouse, September 2009

[4] A. Liu et M. Gouda, “Diverse Firewall Design,” IEEE Transactions on parallel and distributed systems, vol. 19, no. 8, August 2008

[5] A. Liu et M. Gouda, “Structured Firewall Design,” Computer Networks: The International Journal of Computer and Telecommunications Networking, Volume 51, Issue 4, Pages 1106-1120, Mars 2007

[6] E. Al-Shaer et H. Hamed, “Modeling and Management of Firewall Policies,” IEEE Transactions on Network and Service Management, Volume 1-1, April 2004

[7] K. Karoui, F. B. Ftima, and H. B. Ghezala, “Formal Specification, Verification and Correction of Security Policies Based on the Decision Tree Approach,” International Journal of Data & Network Security, vol. 3, no. 3, pp. 92–111, August 2013.

[8] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, P. Mohapatra, “FIREMAN: A Toolkit for FIREwall Modeling and Analysis,” pp.199-213, IEEE Symposium on Security and Privacy, May 2006

[9] R. E. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation,” IEEE Transactions on Computers, vol. 35, no. 8, 1986.

[10] W. Mallouli, J. Orset, A. Cavalli, N. Cuppens, F. Cuppens, “A Formal Approach for Testing Security Rules,” SACMAT'07 Proceedings of the 12th ACM symposium on Access control models and technologies, Sophia Antipolis, France, June 2007.

[11] D. Lee et M. Yannakakis, “Principles and methods of testing finite state machines - A survey,” In Proceedings of the IEEE, volume 84, pages 1090–1126, 1996.

[12] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, G. Trouessin. “OrBAC : un modèle de contrôle d'accès basé sur les organisations,” Cahiers francophones de la recherche en sécurité de l'information, CRIC, University of Montpellier I, n° II, pp.30-43, 2003.

[13] L. Maranget, “Cours de compilation,” Ecole polytechnique, pages 49–66, 2004-2006.

[14] “Transmission Control Protocol, DARPA Internet Program Protocol Specification”, RFC 793, September 1981.

[15] T. Bourdier, “Méthodes algébriques pour la formalisation et l’analyse de politiques de sécurité », PhD thesis, Henri Poincare University, October 2011.