Security Rules Specification and Analysis Based on Passive Testing

Wissam Mallouli[1] Fayçal Bessayah[1] Ana Cavalli[1] and Azzedine Benameur[2]
[1]Institut Telecom SudParis, CNRS/SAMOVAR [2]SAP Research
{wissam.mallouli, faycal.bessayah, ana.cavalli}@it-sudparis.eu
{azzedine.benameur}@sap.com

***Abstract*—Security is a critical issue in dynamic and open distributed environments such as network-based services or wireless networks. To ensure that a certain level of security is maintained in such environments, the system behavior has to be restrained by a security policy in order to regulate the nature and the context of actions that can be performed within the system, according to specific roles. In this paper, we propose a passive testing approach that permits to check whether a system respects its security policy. To reach this goal, we specify this policy using 'Nomad' formal language which is based on deontic and temporal logics. This language is well adapted to passive testing methods that aim to analyze collected system execution traces in order to give a verdict about their conformity with to the system security requirements. Finally, we apply our methodology to an industrial case study provided by SAP group to demonstrate its reliability.**

***Index Terms*—Security Policy Checking, Nomad Language, Trace Collection, Passive Testing.**

I. Introduction

A security policy is a set of rules that defines the desired behavior of users within an information system. Its main goal is to describe how data and other critical system resources are protected. If a security policy is written in a natural language specifying for example: 'file F is only accessible from terminal T in the context C', it will be very difficult to verify its correct implementation using an automatic testing approach because it is a completely informal specification. Consequently, if such verification is not performed, there is no guarantee that the security rules of the system are properly implemented.

Nowadays, security policies are the key point of every modern infrastructure. Challenging security issues concerning network-based services or collaborative applications have arisen because of the complexity and the variety of the implemented system, as well as the high degree of reliability required for their global security. To guarantee that the system respects its security policy, we can rely on formal testing based methods. The main ones are (i) the active testing [17] which validates a system implementation by applying a set of security test cases and analyzing its reaction and (ii) the monitoring (or passive testing) that consists in observing, during the execution, whether the system behavior is conform according to its functional and security formal specification.

In some cases, active testing becomes difficult to perform. This is the case when the tester is provided without any direct interface (called also Observation and Control Point) to interact with the implementation under test (IUT) or when the implementation is built from components that are running in their environment and cannot be shutdown or interrupted for a long period of time. In such situations, there is a particular interest in using passive testing techniques. Indeed, a passive tester does not need to interact with the IUT, it only collects the execution traces and then analyze them without perturbing the IUT behavior. For this reason, we choose to rely in this paper on the passive testing technique to study the conformance of a system according to its security requirements. To perform this analysis, we rely on a dedicated formal language to describe the security requirements of the system. Then, we check using well adapted algorithms whether these security rules are verified on the collected traces to deduce the appropriate verdict about the system security conformance.

Our main contributions in this paper are (i) the formal specification of security policies using a well adapted formalism. This latter is used to describe advanced security rules with temporal aspects regardless of the nature of the tested application. To achieve this, we rely on Nomad language [8] which is more convenient in use than a generic temporal logic (like LTL). (ii) Then, we propose a passive testing approach to analyze collected execution traces, and deduce automatically a verdict concerning the respect of the system to its security policy. We claim that with our methodology and algorithms, all the verdicts are correct: we do not generate any false alarm comparing with previously proposed approaches [2]. (iii) Finally, we present an industrial case study provided by SAP group to demonstrate the reliability of our framework. A discussion of the results is also provided.

The remainder of this paper is organized as follows. In section II, we discuss the related work tackling with the passive monitoring for checking security. Section III presents the formalism we rely on to specify security policies. Our checking methodology is presented in section IV. In section V, we present a case study where SAP R/3 transactions are audited in order to demonstrate the reliability of our testing methodology. Finally, section VI presents the conclusion and introduces the future work.

II. Related Work

Previous work have focused on the definition of languages that allow to specify security policies in a more formal way and verify if there is or conflict [7], [12] between the specified rules. With the great majority of languages, security rules are defined with modalities like permissions, prohibitions and obligations that express possible constraints on the behavior of the system [9]. Among these languages, we can mention for instance Ponder [10] which is an object-oriented language used to describe security rules and management policies in distributed systems, or Or-bac [1] (for Organization Based Access Control) which allows to specify more flexible rules introducing the concept of ‘context’ that allows to describe the circumstances in which certain security rules should be applied. We can also mention Nomad [8] a security model with ‘Non Atomic Actions and Deadlines’ which allows, not only the specification of the rules context, but can also be used to describe temporal constraints in security rules.

Once security policies are formally specified, it remains to verify that the IUT is in conformity with its security policy. Several researches [11], [16], [17] focused on the problem and proposed, on one hand, some approaches based on active testing techniques to generate test suites that can be applied to the system under test. In [17] for example, the authors proposed a framework to specify security policies and to test their implementation on a system. The behavior of the system is specified using the extended finite state machine (EFSM) [15] formalism. The integration of security rules within the system specification is performed using specific algorithms. Then, the automatic tests generation is performed based on a dedicated tool. In [16], the authors adapted model based testing techniques for security policies checking. They proposed an approach to produce test cases from a security policy specified in Or-bac model. To achieve this goal, they first generate test purposes from Or-bac rules then they generate test cases from these test purposes.

On the other hand, some studies [3], [4], [13], [14] rely on passive testing techniques to check the conformance of a system with regards to its functional specification without taking into account security issues. The security checking is usually performed using intrusion detection systems (IDS) that employ either misuse detection or anomaly detection [6].

In this paper, we propose a new methodology based on passive testing techniques to address the problem of the conformance of a system according to its security policy including temporal aspects.

III. Security Policy Specification

A. Formal Language Presentation

To specify the set of security properties that the a system has to respect, we rely on Nomad formal language. The choice of this language was mainly motivated by the characteristics of Nomad that provides a way to describe permissions, prohibitions and obligations related to non-atomic actions within different contexts. Nomad allows to express privileges on non atomic actions. It combines deontic and temporal logics and can describe conditional privileges and obligations with deadlines.

Definition 1: (Atomic action) We define an atomic action as the emission or the reception of a message between two system entities (or components) using the following syntax:

Entity$_1$?or! Msg(Par$_1$,Par$_2$,...,Par$_n$) Entity$_2$

Where: Entity$_1$ and Entity$_2$ represent the source or the destination of the message. ’?’ and ’!’ define a reception and an emission of a message by Entity$_1$. Msg(Par$_1$,Par$_2$,...,Par$_n$) represents the message exchanged between Entity$_1$ and Entity$_2$ with its parameters. Entity$_1$, Entity$_2$, Msg, and Par$_i$ can be replaced by the symbol $*$ to represent any entity, any message or any parameter.

Definition 2: (Non-atomic action) If α and β are actions, then $(\alpha;\beta)$, which means "α is followed immediately by β” and $(\alpha;*;\beta)$, which means " α is followed by β" are non-atomic actions.

Definition 3: (Formula) If α is an action then $start(\alpha)$ (action α is being started) and $done(\alpha)$ (action α is done) are formula.

Here are some properties on actions and formula:

- If A and B are formula then $(A \wedge B)$ and $(A \vee B)$ are formula.
- If A is a formulae then $\neg A$, $\oplus A$ (next in the trace, A will be true), $\oplus^n A$ (in the n next messages in the trace, A will be true), $\ominus A$ (previously in the trace, A was true) and $\ominus^n A$ (in the n previous messages in the trace, A was true) are formula.
- If A is a formula then $O^{\leq d} A$ ($-d$ units of time ago, A was true if $d < 0$ or in the next d units of time, A will be true if $d > 0$) is a formulae.
- $(A|C)$ is a formulae: in the context C the formula A is true.

Definition 4: (Deontic modalities) If A is a formula then modality $\mathcal{O}$ ("A" is mandatory), $\mathcal{F}$ ("A" is forbidden) and $\mathcal{P}$ ("A" is permitted) are formula.

More details about the syntax and semantics of this formal language are presented in [8].

B. Examples of Security Rules Specification

We present in this section some examples of security rules specifications according to Nomad language:

Example 1:

$$\begin{array}{c} \mathcal{P}(\textit{start (usr}_1\textit{! Msg (ReqWrite, fich1.doc) Server}_A\textit{)}| \\ \ominus\textit{(done (usr}_1\textit{ ! Msg (AuthReq) Server}_A\textit{)} \\ \wedge\textit{ done (usr}_1\textit{? Msg (AuthOK) Server}_A\textit{)} \\ \wedge\neg\textit{ done (usr}_1\textit{? Msg (DisconnectReq) Server}_A\textit{)))} \end{array}$$

This rule expresses a permission granted to usr$_1$ to write on ‘file1.doc’ which is managed by Server$_A$, if earlier, the user usr$_1$ was authenticated and not disconnected.

Example 2:

$$\begin{array}{c} \mathcal{O}\textit{(start (Server}_A\textit{! Msg (DisconnectOK) user) }| \\ O^{\leq -30min}\neg\textit{ done (ServerA ? Msg() user))} \end{array}$$

According to this obligation rule, Server$_A$ must disconnect any user if this latter remains inactive for 30 minutes.

Example 3:

$$\begin{array}{c} \mathcal{F}\textit{(start (Server}_A\textit{ ? Msg (https, AuthReq) user) }| \\ O^{\leq -1s}\textit{ done (Server}_A\textit{ ? Msg (https, AuthReq) user} \\ \textit{;*; Server}_A\textit{ ? Msg (https, AuthReq) user))} \end{array}$$

This prohibition rule means that Server$_A$ can not accept more than two authentication requests from the same user in the same second.

The next section presents our passive testing approach to verify security rules specified in Nomad.

IV. Passive Testing Methodology

A. Preliminaries

We can distinguish three steps in our passive testing methodology for security checking:

- The definition of passive testing architecture: in general, to collect execution traces on a running system, we need to install observation points (called also probes) into specific strategic points. These observations points aim to collect data exchanged between relevant entities. The collected traces are usually stored in one or many trace files. In some specific systems (mainly industrial systems), we can have an integrated module within the system that collects all the traces. In both cases, we retrieve trace files that describe the communication between system entities.
- The description of the system security policy using a formal specification language: the description concerns the security rules that the studied system has to respect. We rely in this paper on Nomad language introduced in section III.
- The security analysis: based on the security policy specification, the passive tester has to perform security analysis on the trace file(s) to deduce a global verdict. This verdict is *PASS* if the system trace respects the specified security policy and *FAIL* if it does not. The *INCONCLUSIVE* verdict is possible if the tester can not extract the necessary information from the collected traces in the case of a short trace for example. We assume here that if the trace is long enough (according to the IUT) or if the traffic collection is continuous, we can claim that it describes the global behavior of the system and consequently the verdict concerns the system conformance according to its security policy.

B. Passive Testing Approach

To run the testing process, the security test tool needs two different input files: the trace file collected by the observation point or generated by the system (log file) and a second file where are specified the security policies.

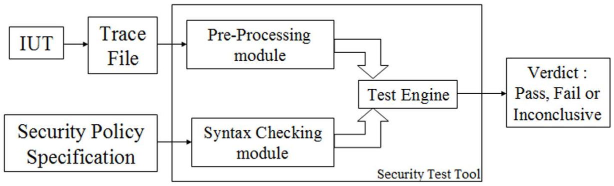


Figure 1. Tool Architecture for Security Checking.

First, the security test tool verifies through a syntax checking module that security policies are well specified according to the Nomad format. This avoids syntax-related bugs in the test engine module.

Second, the collected traces files have to be analyzed using a pre-processing module that performs the following tasks:

- Filtering the traces files keeping only the relevant information for the protocol(s) under test. The basic idea is to keep in the traces only the messages and parameters corresponding to the specified properties to check.
- Parsing the global trace and creating a trace table which constitutes the target of the 'Test Engine' module queries. Each line of the trace table corresponds to an emission or a reception of a message in the network.

Finally, the trace analysis is performed using three algorithms according to the rule nature: permission, prohibition or obligation. These three algorithms are based on the same concept: each line in the trace table can correspond to (i.e. can be an instantiation of) one or many atomic actions described in one or many properties.

Definition 5: (L=Instantiation(A)) a line L in the trace table T is an instantiation of an atomic action A described in the security property Pr if the sender component S, the receiver component R and ***all*** *the message parameters P_i mentioned in the action A are the same existing in the table line L.*

For example, the first line of the table 1 is an instantiation of one of theses actions:
- *Bob ! Msg (AuthReq, 'password = Bob08') Server$_A$*
- *Bob ! Msg (AuthReq, *) Server$_A$*
-   * ! *Msg (AuthReq, 'password = Bob08') Server$_A$*
- *Server$_A$? Msg (AuthReq, *) **

	Msg_Type	Sender	Receiver	Message_Text
L1	AuthReq	Bob	Server$_A$	'password = Bob08'

Table I
An Example of a Line in a Formatted Table

C. Algorithms for Security checking

In this section, we describe the general idea of rules checking algorithms and provide in particular the overview of the algorithm dealing with obligation rules.

1) Obligations Handler: The algorithm that allows checking obligation rules begins first by parsing the trace table line by line to verify the validity of each rule r on a given line l. It begins by verifying the context rule then its mandatory action following these steps: (i) the algorithm verifies if l is an instantiation of an atomic action a mentioned in the context of r. If it is the case, it checks if the chronological order of actions described in the context is verified (using the procedure *Check_Context*), then it can deduce if the whole context is verified or not. (ii) If the context is verified, the algorithm has to ensure that the action described in the first part of the obligation (the mandatory action) is present in the trace. If it finds such action (using *Check_Mandatory_Action* procedure), the verdict is *PASS*. Otherwise, it concludes that the current rule is not verified, the verdict in this case is: *FAIL*. If the trace length is not long enough to ensure the verification, the produced verdict is *INCONCLUSIVE.*

Algorithm 1 presents the pseudo-code of the procedure used to check obligation rules on a trace and deduce the appropriate verdict. For each rule r , we define 'r.action' as its mandatory action and 'r.context' as its context. 'r.action' (resp. 'r.context') is composed of one or many chronologically ordered atomic actions 'r.act.action$_i$' (respectively 'r.context.action$_j$') where i (respectively j) is the number of atomic actions in the prohibited action (respectively context).

Algorithm 1 Obligation Rules Handler

```
Require: $ORS[r]$: Obligation Rules Set + $Tr[l]$: the trace table.
for each Rule $r$ of $ORS$ do
  Context(r) = ‘not verified’
end for
for each line $l$ of $Tr$ do
  for each Rule $r$ of $ORS$ do
    if (Context(r)=‘verified’) then
      verdict[r] := INCONCLUSIVE
      if (Obligation deadline Reached) then
        (a mandatory action has a deadline predefined in the context rule)
        verdict[r] := FAIL
        Memorize error and position in the trace
      else
        if (∃ i where l=instantiation(r.act.action$_i$)) then
          verdict [r] := Check_Mandatory_Action (r.action,l)
          if (verdict [r] := ‘PASS’) then
            Context(r)=‘not verified’
          else
            if (verdict [r] := ‘FAIL’) then
              Memorize error and position in the trace
            else
              Memorize verified parts of the mandatory action (case of verdict [r] := ‘INCONCLUSIVE)
            end if
          end if
        end if
      end if
    end if
    if (∃ j where l=instantiation(r.context.action$_j$)) then
      Context(r) = Check_Context(r.context,l)
      if (Context(r) = ‘verified’) then
        Calculate Obligation deadline
      else
        if (Context(r) = ‘not yet verified’) then
          Memorize verified parts of the context (Context (r) = ‘not yet verified’ if some actions of the context are verified and are in the right chronological order. But the whole context is not yet verified. We have to check next messages in the trace, to deduce if the tested system is in the right context or not.)
        else
          Erase memorized parts of the context if exist (This is case when the context is no more verified)
        end if
      end if
    end if
  end for
end for
```

2) Prohibitions Handler: For prohibitions rules, the approach is very similar to the one used for testing obligations rules. We start first by checking whether the context of the rule is verified. Then, we check if the action specified in the first part of the rule is present in the trace. If it is the case, the verdict is *FAIL* otherwise it is *PASS*. If the trace is not long enough to check the context, the verdict is *INCONCLUSIVE*.

3) Permissions Handler: The permission to perform an action in a particular context does not mean that action must be systematically executed when this context is verified. In the case of checking permission rules, we first look in the traces file the permitted activity then we ensure that the context was true to conclude that the rule is well respected (verdict *PASS*), otherwise the verdict is *FAIL*. If the trace is not long enough to check the context, the verdict is *INCONCLUSIVE*.

V. Case Study: Audit System of SAP R/3

In this section, we apply our methodology on an industrial case study and consider the SAP R/3 technology product of SAP[1] group.

A. System Presentation

SAP R/3 is an enterprise solution developed by SAP group. It is a package that incorporates various functions grouped in distributed modules that can interact with each other through a centralized information system based on a client/server architecture.

SAP R/3 is a real time based system. Thus, every consumption (purchase, sale, etc.) or movement (in stocks for example) has to be immediately valued by updating all system modules involved in this activity. Let us study for example the operation of *a shipment confirmation*. This operation engenders an automatic billing operation, an operation for recording movements in stocks and bills, and possibly other updates at certain accounting services of the company. All these operations have to be done in real time, hence the conception of SAP R/3 as an integrated management software able of updating data of different modules instantaneously.

B. SAP R/3 Security

In SAP R/3, any transaction that could be performed by users is identified by a unique code named: transaction code. For example, the function used to change the *principal supplier* field of the system information table has the transaction code *FK02*, and the code *FB60* is used to describe the transaction, that manages *customer invoices* etc. SAP R/3 administrators have defined a set of rules including permissions, prohibitions and obligations that regulates performing each transaction. However, certain combinations of these transactions codes can lead to situations of conflict and/or incoherence. To avoid such situations, SAP R/3 system uses a generator profile to define generic roles that can be assigned later to users.

In SAP R/3, many security rules are defined to guarantee the system integrity and to control the access to critical data. We identified more than 120 different rules relatives to 10 clients and 26 possible operations (see section V-D).

[1] http://www.sap.com

	Date	Time	Client	User	Code	Terminal	ID	Message Text
E1	01.04.2007	08:55:04	600	Bob		S826-01	AU2	Login Failed
E1	01.04.2007	08:56:30	600	Bob		S826-01	AU1	Login Successful
E2	01.04.2007	13:43:11	600	Bob	F110	S826-01	AU4	Transaction F110 Failed
E3	08.04.2007	08:55:04	654	John		S826-01	AU2	Login Failed
E3	08.04.2007	08:55:06	654	John		S826-01	AU2	Login Failed
E3	08.04.2007	08:55:08	654	John		S826-01	AU2	Login Failed
E3	08.04.2007	08:55:09	654	John			AUM	User John Locked in Client 654

Table II
EXAMPLE OF FORMATTED CONTENTS IN A FILE AUDIT

C. *Trace Collection: Audit System*

SAP R/3 has an audit system that permits to store all the events and transactions that occur during a period of time and to give the system administrators an overview about the users' activities within the system. It provides a set of files that contain detailed information about every activity undertaken by system users. These files represent the execution traces of SAP R/3 and constitute an important source of data for all kinds of testing or checking system security to detect any intrusion attempts, fraud, or any other malicious activity. In our case, we will use these files to test the respect of SAP R/3 to its security policy specified by administrators.

In our case study and for confidentiality reasons, we collected the execution traces of a running SAP R/3 system managing a fake database (with fake users' names and fake transactions). All the transactions are performed randomly and many times by all the users. The obtained log file has a size of 800 MByte and contains the details of 2.5 millions transactions (lines). Each transaction has a pre-formatted structure which can be divided into 8 fields.

In Table II, we present three formatted examples of events that we can find in a file log of SAP R/3. In the first one (E1), the user *'Bob'* connects from the workstation 1 which is located in Room S826. In second event (E2), the same user tried unsuccessfully to execute the transaction F110 which deals with a payment operation. In E3, the user *'John'* has made three attempts to connect without success. Hence this account has been locked by the system.

D. *Security Policy Formal Specification*

Based on nomad language, we specified formally the 120 security rules. For matter of space, only 3 basic rules are presented in this section:

- Rule 1: $\mathcal{P}(start(John\ !\ Msg(Tr.Code = FK01)\ R/3))$

This rule means that *'John'* is permitted to execute the transaction FK01.

- Rule 2: $\mathcal{F}(start(User\ !\ Msg(*)\ R/3)|$
$\ominus\, done(User\ ?\ Msg(ID = AUM)\ R/3) \wedge$
$\neg done(User\ ?\ Msg(ID = UNLOCK)\ R/3))$

This rule expresses that any user whose access has been locked by the system is prohibited to perform transactions.

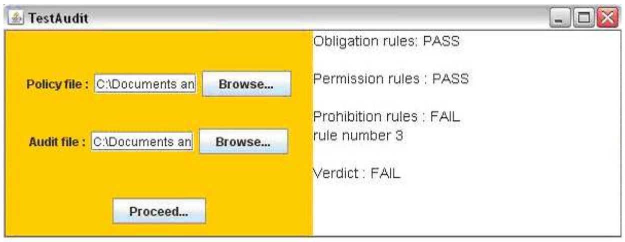
- Rule 3: $\mathcal{O}(Start(User\ ?\ Msg(ID = AUM)\ R/3)| \ominus done((User\ !\ Msg(ID = AU2)\ R/3); (User\ !\ Msg(ID = AU2)\ R/3); (User\ !\ Msg(ID = AU2)\ R/3)))$

This obligation rule expresses that the R/3 system has to lock any user who has made three unsuccessful connection attempts.

E. *Passive Tester Implementation*

The passive tester dedicated to monitoring SAP R/3 system respects the design presented in section IV-B. This tool is entirely programmed using Java language and is designed to offline passive testing.The extraction of the traces file is not taken into account in this passive tester tool.

We can distinguish in the graphical interface of the tool, presented in Figure 2, three different features: (i) *'A Policy File Browse Button'* that allows to upload the security rules specified using nomad language (ii) *'An Audit File Browse Button'* that allows to upload the collected traces of SAP R/3 execution. (iii) *'A Blank Space'* that allows to display the result of the security checking and to give the final verdict and eventually the violated rule.



Obligation rules: PASS

Permission rules : PASS

Prohibition rules : FAIL
rule number 3

Verdict : FAIL

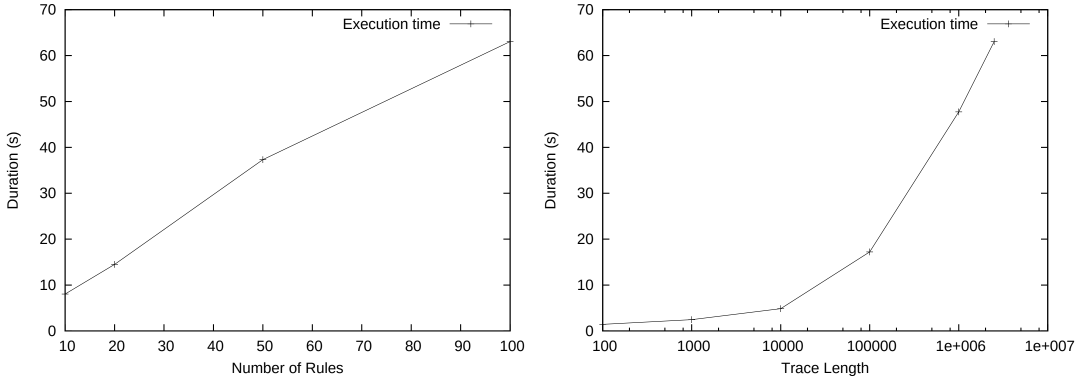
Figure 2. Passive Tester Interface in the Case of a Fail Verdict

F. *Passive Testing Results*

We first performed our test to check the 120 rules on the SAP R/3 collected traces. The result of this first test was a PASS verdict which means that according to information contained in the audit file, all security rules were respected (we checked manually that each rule was verified at least once). To obtain this final verdict, the tester performed a simple Boolean operation, which combines the three partial verdicts already established by the three different sub-modules dedicated for permissions, prohibitions and obligations. Thus, if one of these verdicts is INCONCLUSIVE or FAIL the final verdict will be too. Otherwise PASS is the verdict to be deduced.

To demonstrate the reliability of our tester, we manually edited the file containing traces of SAP R/3 system. We have, for instance, added a transaction that violates the rule number 3. The verdict given by the tester shown in Figure 2 is FAIL: the violation of the rule has been detected which indicates the correctness of the tester.

To study the performances of the passive tester, we tried to vary the number of security rules and the Traces file size. The results are shown in Figures 3.



(a) Tool performance with a variable number of Rules

(b) Tool Performance with a variable audit file length

Figure 3. Tool Performances

In Figure 3.a, we vary the number of security rules from 5 to 120 and we fix the audit file length to 2.5 millions transactions. We obviously observe that the curve is growing. The curve is not linear; the slope of the curve depends on the complexity of the added rules. The memory consumption increases and leads to bigger time to deduce a verdict.

Contrary to Figure 3.a, in Figure 3.b, we vary the audit file length and we consider that the number of rules is fixed to 100. The curve is also growing in non linear manner. This result is predictable and is due to a bigger complexity to verify a rule context. The memory consumption also increases with the number of lines in the audit trace and leads to bigger duration to deduce a verdict. When the trace size becomes bigger than one million lines, the execution time becomes exponential due to memory swapping but remains reasonable.

VI. Conclusions and Future Work

In this paper, we presented a security policies specification model well adapted for passive testing. We first began by defining the syntax and the semantics of the proposed formalism and illustrated it with a few examples of specifications. The next step was to expose our passive testing methodology and to demonstrate its effectiveness, through an industrial case study proposed by SAP group namely: SAP R/3.

Finally, at the end of this paper, we presented the experimental results we had obtained and discussed about the performance of the implemented tester. It is important to notice that the performances of our tester are very suitable compared to the complexity of the described rules and the length of the trace. We also showed that our approach permitted us to specify and verify temporal security constraints that are considered as an important issue in security policies testing.

As future work, we plan to adapt the proposed model for a specific type of applications such as Web services and authentication protocols. We are also investigating several approaches to improve the test algorithms so that we will be able to perform online passive testing, possibly by including vulnerability cause graphs [5] of the implementation under test. This will enable us to detect on real-time system crashes and security rules violations and most importantly to be able to stop this kind of malicious behaviors without any delay.

References

[1] A. Abou El Kalam, R. E. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'03)*, June 2003.

[2] A. Alharby and H. Imai. IDS false alarm reduction using continuous and discontinuous patterns. In *ACNS*, pages 192–205, 2005.

[3] E. Bayse, A. R. Cavalli, M. Núñez, and F. Zaïdi. A passive testing approach based on invariants: application to the wap. *Computer Networks*, 48(2):235–245, 2005.

[4] A. Benharref, R. Dssouli, R. H. Glitho, and M. A. Serhani. Towards the testing of composed web services in 3rd generation networks. In *TestCom*, pages 118–133, 2006.

[5] D. Byers, S. Ardi, N. Shahmehri, and C. Duma. Modeling software vulnerabilities with vulnerability cause graphs. In *ICSM*, pages 411–422, 2006.

[6] Y. Chen, Y. Li, X. Cheng, and L. Guo. Survey and taxonomy of feature selection algorithms in intrusion detection system. In *Inscrypt*, pages 153–167, 2006.

[7] F. Cuppens, N. Cuppens-Boulahia, and M. B. Ghorbel. High level conflict management strategies in advanced access control models. *Electr. Notes Theor. Comput. Sci.*, 186:3–26, 2007.

[8] F. Cuppens, N. Cuppens-Boulahia, and T. Sans. Nomad: A security model with non atomic actions and deadlines. In *CSFW*, pages 186–196, 2005.

[9] N. Damiannnou, A. Bandara, M. Sloman, and E. Lupu. *Handbook of Network and System Administration*, chapter A Survey of Policy Specification Approaches. Elsevier, 2007 (to appear).

[10] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. Ponder: An object-oriented language for specifying security and management policies. In *10th Workshop for PhD Students in Object-Oriented Systems (PhDOOS'2000), 12-13 June 2000, Sophia Antipolis, France*, 2000. Extended Abstract.

[11] V. Darmaillacq, J.-C. Fernandez, R. Groz, L. Mounier, and J.-L. Richier. Test generation for network security rules. In *TestCom*, pages 341–356, 2006.

[12] J. García-Alfaro, F. Cuppens, and N. Cuppens-Boulahia. Analysis of policy anomalies on distributed network security setups. In *ESORICS*, pages 496–511, 2006.

[13] H. Hallal, S. Boroday, A. Ulrich, and A. Petrenko. An automata-based approach to property testing in event traces. In *TestCom*, pages 180–196, 2003.

[14] T. Jéron, H. Marchand, S. Pinchinat, and M.-O. Cordier. Supervision patterns in discrete event systems diagnosis. Technical Report 1784, Irisa, 2006.

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

[16] K. Li, L. Mounier, and R. Groz. Test generation from security policies specified in or-bac. In *COMPSAC (2)*, pages 255–260, 2007.

[17] W. Mallouli, J.-M. Orset, A. R. Cavalli, N. Cuppens-Boulahia, and F. Cuppens. A formal approach for testing security rules. In *SACMAT*, pages 127–132, 2007.