11

A Guided Web Service Security Testing Method

Sébastien Salva
LIMOS UMR CNRS 6158, University of Auvergne
France

1. Introduction

For the last five years, the Internet is being revolutionized by becoming a Service-oriented platform. This tremendous inflection point in Computer Science leads to many new features in design and development such as the deployment of interoperable services accessible from Web sites or standard applications, the modelling of high level Business processes orchestrating Web Service sets, or recently the virtualization of service-based applications by means of the Cloud paradigm.

To achieve reliable Web services, which can be integrated into compositions or consumed without any risk in an open network like the Internet, more and more software development companies rely on software engineering, on quality processes, and quite obviously on testing activities. In particular, security testing approaches help to detect vulnerabilities in Web services in order to make them trustworthy. Nevertheless, it is quite surprising to notice that few security testing methods have been proposed for Web Services. This chapter addresses this issue by presenting a formal security testing method for stateful Web Services. Such services are persistent through a session and have an internal state which evolves over operation call sequences. For instance, all the Web Services using shopping carts or beginning with a login step are stateful. The proposed method aims to experiment *black box* Web Services, from which only SOAP messages (requests and responses) are observable. We do not have access to the code, Web services can be experimented only through their interfaces. Our approach is an active Model Based one: it relies on a specification formalized with a model to test Web services by means of test cases generated from the model. Model based testing approaches offer many advantages such as the description of a service without ambiguity. Accompanied with a formal method, some steps of the test can be also automated, e.g., the test case generation Rusu et al. (2005); Tretmans (2008). The use of a model also helps to define a relation between the specification and its black-box implementation to express clearly the confidence level between them. In this paper, we model Web services with Symbolic Transition Systems (STS Frantzen et al. (2005)) describing the different states, the called operations and the associated data.

In literature, for the same reasons, security policies are often described by means of formal rules, which regulate the nature and the context of actions that can be performed. Several security rule languages have been introduced in Cuppens et al. (2005); Senn et al. (2005). We have chosen Nomad (Non atomic actions and deadlines Cuppens et al. (2005)) to model abstract test patterns which can be directly derived from an existing security rule set. Nomad is well suited for expressing properties such as permissions, prohibitions or obligations and

is able to take into account response delays. Our approach takes a Web Service specification and applies abstract test patterns on the operation set to generate test requirements called test purposes. These ones, which guide the tests, are then synchronized with the specification to produce the test case suite. The latter checks the satisfiability of the test relation *secure*, which formally defines the security level between the implementation and its specification combined with test purposes. The Amazon E-commerce Web Service (AWSECommerceService) Amazon (2009) is illustrated as an example on which we apply our method.

Another part of the book chapter is dedicated to the experimentation of the method on existing Web Services with an academic tool. The obtained results demonstrate a dramatic lack of security in many Web Services since 11 percent dot not satisfy the restrictions given by our security test patterns.

This book chapter is structured as follows: Section 2 provides an overview of the Web Service paradigm and on some related works about Web Service security testing. Sections 3 and 4 describe the Web service and test pattern modelling respectively. The testing method is detailed in Section 5. In Section 6, we discuss some experimentation results, the test coverage and the complexity of the method. Finally, Section 7 gives some perspectives and conclusions.

2. Web services security overview

Web Services are *"self contained, self-describing modular applications that can be published, located, and invoked across the Web"* Tidwell (2000). To ensure the Web Service interoperability, the WS-I organization has suggested profiles, and especially the WS-I basic profile WS-I organization (2006), composed of four major axes: the Web Service interface description with the WSDL language (Web Services Description Language World Wide Web Consortium (2001)), the definition and the construction of XML messages, based upon the Simple Object Access Protocol (SOAP World Wide Web consortium (2003)), the service discovery in UDDI registers (Universal Description, Discovery Integration Specification (2002)), and the Web service security, which is obtained by using the HTTPS protocol.

It is surprising to notice that security was the poor relation during the rush to Web Services and it is manifest that the HTTPS protocol was not sufficient to fulfill the security requirements of service-based applications. We can now find a tremendous set of documents and specifications related to Service security. The WS-security standard (Web Service Security OASIS consortium (2004)) gathers most of them. This document describes a SOAP rich extension to apply security to Web services by bringing message encryptions, message signing, security token attachment, etc. Both the policy requirements of the server side and the policy capability of the client side can be expressed by means of the WS-Policy specification. Nevertheless, this one is "only" SOAP-based, and defines requirements on encryption, signing or token mechanisms. Higher level rules cannot be expressed with WS-Policy.

Besides these specifications, several academic papers Gruschka & Luttenberger (2006); ISO/IEC (2009); Singh & Pattterh (2010) and the OWASP organization OWASP (2003) focused on Service security in regard to access control by decomposing it into several criteria: availability, integrity, confidentiality, authorization, authentication and freshness and by proposing recommendations for each one. Each criterion can be also modelled formally by means of security rules written with languages such as XACML (eXtensible Access Control Markup Language OASIS standards organization (2009)), Nomad (Security Model with Non

Atomic Actions and Deadlines Cuppens et al. (2005)), or OrBAC (Organisation-based access control Kalam et al. (2003)).

Some other papers, focusing on security rule modelling and formal testing, have been proposed in literature. Modelling specifications for testing and defining formal methods is more and more considered in literature and in industry because this offers many advantages such as the definition of the confidence level of the implementation in comparison with its specification, the coverage of the tests, or the automation of some steps. An overview of model based testing is given in Tretmans (2008).

These works can be grouped into the following categories:

- *Test Generation for Model-based Policies*. Test generation methods for model-based policies construct abstract test cases directly from models describing policies. For instance, Le Traon et al. Le Traon et al. (2007) proposed test generation techniques to cover security rules modelled with OrBAC. They identified rules from the policy specification and generated abstract test cases to validate some of them. Senn et al. showed, in Senn et al. (2005), how to formally specify high-level network security policies, and how to automatically generate test cases, by using the specification. In Darmaillacq et al. (2006), network security rules are tested by modelling the network behaviour with labelled transition systems. Then, test patterns are injected into existing test cases to validate the rules.
- *Random Test Generation:* or fuzzy testing is a technique which automatically or semi-automatically constructs test cases with random values. For instance, in Martin (2006), the authors developed an approach for random test generation from XACML policies. The policy is analyzed to generate test cases by randomly selecting requests from the set of all possible requests,
- *Mutation testing:* usually involve mutation of policies or programs. In Mouelhi et al. (2008), the authors proposed a model-driven approach for specifying testing security policies in Java applications. The policy is modelled with a control language such as OrBAC and translated into XACML. Then, the policy is integrated into the application. Faults are injected into the policy to validate the policy in the application by mutating the original security rules.

Concerning, the Web service security testing, which is the topic of the chapter, few dedicated works have been proposed. In Gruschka & Luttenberger (2006), the passive method, based on a monitoring technique, aims to filter out the SOAP messages by detecting the malicious ones to improve the Web Service's availability. Mallouli et al. also proposed, in Mallouli et al. (2008), a passive testing method which analyzes SOAP messages with XML sniffers to check whether a system respects a policy. In Mallouli et al. (2009), a security testing method is described to test systems with timed security rules modelled with Nomad. The specification is augmented by means of specific algorithms for basic prohibition and obligation rules only. Then, test cases are generated with the "TestGenIF" tool. A Web Service is illustrated as an example.

Our first motivation comes from the paper Mallouli et al. (2009) which describes a testing method from Nomad rules. This one can handle basic rules composed of abstract actions only and can be applied on generic systems. In this book chapter, we intend to propose a specific security testing method which takes into account black box Web Services deployed in a SOAP

environment that is used to invoke operations. We claim that SOAP must be considered while testing since it modifies the Web Service behaviour and thus may falsify the testing verdict. So, in Section 3.2 we study the Web service consuming with SOAP and propose a specification completion to take it into account in the test case generation.

We consider the access control-based vulnerabilities expressed in OWASP (2003) to describe some security test patterns composed of actions such as operation requests. Actually, we specialize our test patterns for Web services to experiment our method. So, test patterns are composed of malicious requests (XML and SQL injections) for testing the Web Service availability, authentication and authorization. They also contain different variable domain sets such as RV composed of values well-known for detecting bugs and random values, or Inj composed of values for both SQL and XML injections. So, our method covers several categories cited previously (model-based and random testing). Then, we present a dedicated testing methodology, based upon a formal security test relation, denoted *secure* which expresses, without ambiguity, the security level of an implementation with regard to its specification and to a set of test patterns. To check the satisfiability of *secure*, test patterns are translated into test purposes. Then, concrete test cases are generated by means of a synchronous product between the specification and test purposes. Intuitively, we obtain action sequences, extracted from the specification and also composed of the initial test pattern properties. Our test purpose-based method helps to reduce the specification exploration during the test case generation and thus reduces the test costs Castanet et al. (1998).

Prior to present the testing methodology, we define the Web service modelling below.

3. Web Service modelling in SOAP environments

Several models e.g., UML, Petri nets, process algebra, abstract state machines (ASM), have been proposed to formalize Web services. STSs (Symbolic Transition Systems Frantzen et al. (2005)) have been also used with different testing methods Frantzen et al. (2006); ir. H.M. Bijl van der et al. (2003); Salva & Rabhi (2010). The STS formalism offers also a large formal background (process algebra notations, definitions of implementation relations, test case generation algorithms, etc.). So, it sounds natural to use it for modelling specifications and test cases. Below, we recall the background of the STS formalism and the specification completion to take into account the SOAP environment.

3.1 Stateful Web Service modelling

An STS is a kind of input/output automaton extended with a set of variables, with guards and assignments on variables labelled on the transitions. The action set is separated with inputs beginning by ? to express the actions expected by the system, and with outputs beginning by ! to express actions produced (observed) by the system. Inputs of a system can only interact with outputs provided by the system environment and vice-versa.

Definition 1. *A Symbolic Transition System STS is a tuple* $< L, l_0, V, V_0, I, \Lambda, \rightarrow >$*, where:*

- *L is the finite set of locations, with l_0 the initial one,*
- *V is the finite set of internal variables, while I is the finite set of external or interaction ones. We denote D_v the domain in which a variable v takes values. The internal variables are initialized with the assignment V_0, which is assumed to take an unique value in D_V,*

- Λ *is the finite set of symbols, partitioned by* $\Lambda = \Lambda^I \cup \Lambda^O$*: inputs, beginning with ?, are provided to the system, while outputs (beginning with !) are observed from it.* $a(p) \in \Lambda \times I^n_{n\geq 0}$ *is an action where* p *is a finite set of parameters* $p = (p_1, ..., p_k)$*. We denote* $type(p) = (t_1, ..., t_k)$ *the type of the variable set* p*, and* D_p *the variable domain in which* p *takes values,*
- $\rightarrow$ *is the finite transition set. A transition* $(l_i, l_j, a(p), \varphi, \varrho)$*, from the location* $l_i \in L$ *to* $l_j \in L$*, also denoted* $l_i \xrightarrow{a(p),\varphi,\varrho} l_j$ *is labelled by* $a(p) \in \Lambda \times I^n_{n\geq 0}$*,* $\varphi \subseteq D_V \times D_p$ *is a guard which restricts the firing of the transition. Internal variables are updated with the assignment* $\varrho : D_V \times D_p \rightarrow D_V$ *once the transition is fired.*

The STS model is not specifically dedicated (restricted) to Web services. This is why we assume that an action $a(p)$ represents either the invocation of an operation op which is denoted $opReq$ or the return of an operation with $opResp$. Furthermore, Web service are object-oriented components which may throw exceptions. So, we also model exception messages with a particular symbol denoted $!exp \in \Lambda^O$.

For simplicity and to respect the WS-basic profile, we assume that operations either always return a response or never (operations cannot be overloaded). We also assume that STSs are deterministic. As a consequence, we suppose that operations are synchronous, i.e. these ones return a response immediately or do nothing. Asynchronous methods may return a response anytime, from several states and often imply indeterminism.

An immediate STS extension is called the STS *suspension* which also expresses quiescence i.e., the absence of observation from a location. Quiescence is expressed with a new symbol $!\delta$ and an augmented STS denoted $\Delta(STS)$. For an STS $\mathcal{S}$, $\Delta(\mathcal{S})$ is obtained by adding a self-loop labelled by $!\delta$ for each location where quiescence may be observed. The guard of this new transition must return true for each value of $D_{V \cup I}$ which does not allow firing a transition labelled by an output.

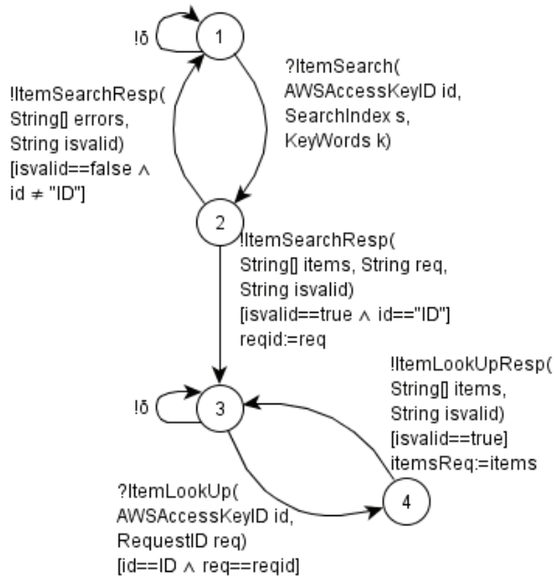


Fig. 1. An STS specification

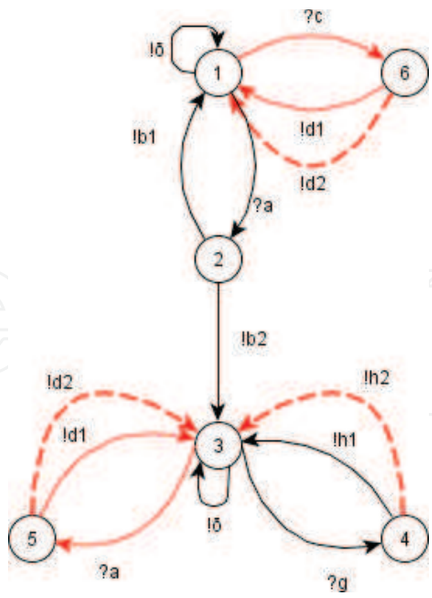


Fig. 2. The completed specification

A specification example, is illustrated in Figure 1. This one describes, without ambiguity, a part of the Amazon Web Service devoted for e-commerce (AWSECommerceService Amazon (2009)). For simplicity, we consider only two operations: *ItemSearch* aims to search for items, and *ItemLookUp* provides more details about an item. The AWSAccessKeyID parameter uniquely identifies the user of the Web service and is provided by Amazon. The SearchIndex parameter is used to identify the item in demand. "book" is a classical value. Notice that we do not include all the parameters for readability reasons.

An STS is also associated to an LTS (Labelled Transition System) to define its semantics. Intuitively, the LTS semantics represents all the functional behaviours of a system and corresponds to a valued automaton without symbolic variables: the states are labelled by internal variable values while transitions are labelled with actions and parameter values.

Definition 2. *The semantics of an STS* $\mathcal{S} =< L, l_0, V, V_0, I, \Lambda, \rightarrow >$ *is an LTS* $||\mathcal{S}|| =< Q, q_0, \sum, \rightarrow >$ *where:*

- $Q = S \times D_V$ *is the finite set of states,*
- $q_0 = (l_0, V_0)$ *is the initial state,*
- $\sum = \{(a(p), \theta) \mid a(p) \in \Lambda, \theta \in D_p\}$ *is the set of valued symbols,*
- $\rightarrow$ *is the transition relation* $S \times \Sigma \times S$ *deduced by the following rule:*

$$\frac{l_i \xrightarrow{a(p),\varphi,\varrho} l_j, \theta \in D_p, v \in D_V, v' \in D_V, \varphi(v,\theta)\ true, v' = \rho(v,\theta)}{(l_i, v) \xrightarrow{a(p),\theta} (l_j, v')}$$

This rule can be intuitively read as follows: for an STS transition $l_i \xrightarrow{a(p),\varphi,\varrho} l_j$, we obtain a LTS transition $(l_i, v) \xrightarrow{a(p),\theta} (l_j, v')$ with v an internal variable value set, if it exists a parameter value θ such that the guard $\varphi(v, \theta)$ is satisfied. Once the transition is executed, the internal

variables take the value v' derived from the assignment $\varrho(v, \theta)$. An STS suspension $\Delta(\mathcal{S})$ is associated to its LTS semantics suspension by $||\Delta(\mathcal{S})|| = \Delta(||\mathcal{S}||)$.

Some behavioural properties can now be defined on STS in terms of their underlying semantics, in particular runs and traces.

Definition 3 (Runs and traces)**.** *For an STS $\mathcal{S}$, interpreted by its LTS semantics $||\mathcal{S}|| =< Q, q_0, \sum, \rightarrow >$, a run $q_0\alpha_0...\alpha_{n-1}q_n$ is an alternate sequence of states and valued actions. $RUN(\mathcal{S}) = RUN(||\mathcal{S}||)$ is the set of runs found in $||\mathcal{S}||$. $RUN_F(\mathcal{S})$ is the set of runs of $\mathcal{S}$ finished by a state in $F \subseteq Q$.*
It follows that a trace of a run r is defined as the projection $proj_{\sum}(r)$ on actions. So, $Traces_F(\mathcal{S}) = Traces_F(||\mathcal{S}||)$ is the set of traces of runs finished by states in $F \subseteq Q$.

The traces of a STS suspension $Traces_F(\Delta(\mathcal{S}))$ also called the suspension traces are denoted $STraces_F(\mathcal{S})$.

3.2 The SOAP environment

Web services are deployed in specific environments, e.g., SOAP or REST, to structure messages in an interoperable manner and/or to manage operation invocations. Such environments may modify the observable reactions of a Web service implementation, for instance by adding and modifying the requests and responses. These modifications must be taken into account in testing methods to ensure that the test verdict is not falsified by the environment.

In this book chapter, we consider the SOAP environment only: it consists in a SOAP layer which serializes messages with XML and of SOAP receivers (SOAP processor + Web services) which is a software, in Web servers, that consumes messages (WS-I organization (2006)). The SOAP processor is a Web service framework part which represents an intermediary between client applications and Web services and which serializes/deserializes data and calls the corresponding operations. We summarize below the significant modifications involved by SOAP processors:

- **Calling an operation which does not exist:** this action produces the receipt of a response, constructed by Soap processors, which corresponds to a Soap fault composed of the cause "the endpoint reference is not found". Soap faults are specific XML messages which give details e.g., a cause (reason) about a triggered exception or a crash,
- **Calling an existing operation with incorrect parameter types:** this action produces also the receipt of a Soap fault, constructed by the Soap processor, composed of the cause "Client". This one means that the Client request does not match the Web Service WSDL description,
- **Exception management:** by referring to the WS-I basic profile WS-I organization (2006), when an exception is triggered by a Web Service operation, then the exception ought to be translated into a Soap fault and sent to the Client application. However, this feature needs to be implemented by hands in the operation code. So, when the exception management is implemented, the Soap fault cause is usually equal to "SoapFaultException" (in Java or C# implementations). Otherwise, the operation crashes and the Soap processor may construct itself a Soap fault (or do nothing, depending on the chosen Web Service framework). In this case, the Soap fault cause is different from "SoapFaultException".

In summary, SOAP processors add new messages, called SOAP faults, which give details about faults raised in the server side. They return SOAP faults composed of the causes "Client" or "the endpoint reference not found" if services or operations or parameter types do not exit. SOAP processors also generate SOAP faults when a service instance has crashed while triggering exceptions. In this case, the fault cause is equal to the exception name. However, exceptions correctly managed in the specification and in the service code (with try...catch blocks) are distinguished from the previous ones since a correct exception handling produces SOAP faults composed of the cause "SOAPFaultException". Consequently, it is manifest that SOAP modifies the behaviour of the specification by adding new reactions which must be taken into account while testing.

So, we propose to augment an initial specification with the SOAP faults generated by SOAP processors. We denote $!soapfault(cause)$ a SOAP fault where the external variable *cause* is the reason of the SOAP fault receipt.

Let $\mathcal{S} =< L_{\mathcal{S}}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{\mathcal{S}}, \rightarrow_{\mathcal{S}} >$ be an STS and $\Delta(\mathcal{S})$ be its suspension. $\Delta(\mathcal{S})$ is completed by means of the STS operation *addsoap* in $\Delta(\mathcal{S})$ which augments the specification with SOAP faults as described previously. The result is an STS $\mathcal{S}^{\uparrow}$. The operation *addsoap* is defined as follow: *addsoap* in $\Delta(\mathcal{S}) =_{def} S^{\uparrow} =< L_{\mathcal{S}\uparrow}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}\uparrow}, \Lambda_{\mathcal{S}\uparrow}, \rightarrow_{\mathcal{S}\uparrow} >$ where $L_{\mathcal{S}\uparrow}$, $I_{\mathcal{S}\uparrow}$, $\Lambda_{\mathcal{S}\uparrow}$ and $\rightarrow_{\mathcal{S}\uparrow}$ are defined by the following rules:

Exception Soapfault: to $\dfrac{l_1 \xrightarrow{!exp,\varphi,\varrho}_{\Delta(\mathcal{S})} l_2}{l_1 \xrightarrow{!soapfault(c),\varphi,\varrho}_{\mathcal{S}\uparrow} l_2}$

Input completion: $\dfrac{l_1 \xrightarrow{?opReq(p),\varphi,\varrho}_{\Delta(\mathcal{S})} l_2, l_1 \xrightarrow{?op_iReq(p),\varphi',\varrho'} l \notin \rightarrow_{\Delta(\mathcal{S})}, ?opReq_i \in \Lambda^I_{\Delta(\mathcal{S})}, l' \notin L_{\Delta(\mathcal{S})}}{l_1 \xrightarrow{?op_iReq(p),\emptyset,\emptyset}_{\mathcal{S}\uparrow} l', l' \xrightarrow{!soapfault(c)),\varphi=[c\neq"CLIENT" \wedge c\neq"the\ endpoint..."],\emptyset}_{\mathcal{S}\uparrow} l_1}$

SOAPfault completion: $\dfrac{l \xrightarrow{?opReq(p),\varphi,\varrho}_{\Delta(\mathcal{S})} l', \varphi' = \bigwedge \neg\varphi \quad l' \xrightarrow{!op_iResp(p),\varphi,\varrho}_{\Delta(\mathcal{S})} l''}{l' \xrightarrow{!soapfault(c),\varphi',\emptyset}_{\mathcal{S}\uparrow} l}$

The first rule translates an exception message into a SOAPfault. The second one completes the specification to be input enabled. Indeed, it is stated, in the WS-I basic profile WS-I organization (2006), that any Web service operation can be invoked anytime. So, we assume that each unspecified operation request should return a SOAP fault message. The last rule completes the output set by adding, after each transition modelling an operation request, a transition labelled by a SOAP fault. Its guard corresponds to the negation of the guards of transitions modelling responses. This transition refers to the exception management. When any exception is triggered in the server side, a SOAP fault is sent.

A completed specification example is illustrated in Figure 2 where the solid red transitions represent the operation call completion and the dashed ones the SOAP fault completion. The symbol table is given in Figure 3. For instance, the transition from location 4 labelled by !h2 is added to express that after calling the operation $ItemLookUp$ a SOAP fault may be received.

Symb	Message	Guard	Update
?a	?ItemSearchReq(AWSAccessKeyID id,SearchIndex s, KeyWords k)		
?atp	?ItemSearchReq(AWSAccessKeyID id,SearchIndex s, KeyWords k)		TestDom:={Spec(ItemSearch);RV;Inj}
!b1	!ItemSearchResp(String[] errors, String isvalid)	[isvalid==false ∨ id<>"ID"]	
!b2	!ItemSearchResp(String[] items, String req, String isvalid)	[isvalid==true ∧ id=="ID"]	reqid:=req
?c	?ItemLookUpReq(AWSAccessKey ID id, RequestID req)		
!d1	!soapfault(c)	[c≠"Client" ∧ c≠"the endpoint..."]	
!d2	!soapfault(c)	[c=="Client" ∨ c=="the endpoint..."]	
!ftp	!soapfault(c)	[c=="SOAPFaultException"]	
?g	?ItemLookUpReq(AWSAccessKey ID id, RequestID req)	[id==ID ∧ req==reqid]	
!h1	!ItemLookUpResp(String[] items, String isvalid)	[isvalid==true]	itemsReq:= items
!h2	!soapfault(c)	[isvalid≠true]	

Fig. 3. Symbol table

4. Nomad test patterns

Security policies are more and more expressed by means of formal languages to express without ambiguity concepts such as obligation, permission or prohibition for an organization. A natural way to test policies consists in deriving, manually or semi-automatically, test cases directly from the latter. Usually, we obtain abstract tests, that we call *test patterns*. Some works Mouelhi et al. (2008) have proposed solutions to derive test patterns from basic security rules.

In this Section, to illustrate our methodology and to experiment existing Web services, we propose to formalize some test patterns from the recommendations provided by the OWASP organization OWASP (2003). Thereby, these test patters are specialized for Web services and will help to detect attacks/vulnerabilities such as empty passwords, brute force attack, etc. They are related to the following criteria: availability, authentication and authorization. We do not provide an exhaustive test pattern list, because this one depends firstly of the security policy established by an organization and because an exhaustive list would deserve a separate book of its own. Nevertheless, the following test patterns cover most of the OWASP recommendations. These test patterns are constructed over a set of attacks (brute force, SQL injection, etc.) and model how a Web service should behave when it receives one of them.

As stated previously, we have chosen to formalize test patterns with the Nomad language. Nomad is based upon a temporal logic, extended with alethic and deontic modalities. It can easily express the obligation, the prohibition and the permission for atomic or non-atomic actions with eventually timed constraints. Bellow, we recall a part of the Nomad grammar. The complete definition of the Nomad language can be found in Cuppens et al. (2005).

Nomad notations:
- If A and B are actions, then $(A;B)$ (A followed by B) and (A & B) (A in parallel with B) are actions
- If A is an action then $start(A)$, $doing(A)$, and $done(A)$ are formulae
- If α and β are formulae, then $\neg\alpha$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, and $(\alpha \Leftrightarrow \beta)$ are formulae
- If α is a formulae, then $\mathcal{O}\alpha$ (α is obligatory), $\mathcal{F}\alpha$ (α is forbidden"), $\mathcal{P}\alpha$ (α is permitted) are formulae
- $\mathcal{O}^{\leq d}A$ represents an obligation with deadline, and is to be read: "it is obligatory that A within a delay of d units of time"
- The last definition concerns the conditional privilege: if α and β are formulae, $(\alpha|\beta)$ is a formula whose semantic is "in the context β, α is true"

As a first step, we augment the Nomad language with this straightforward expression to model the repeating of an action:

If A is an action, then $A^n = A;...;A$ (n times) is an action.

Now, we are ready for the test pattern description.

4.1 Availability test patterns

The Web Service availability represents its capability to respond correctly whatever the request sent. Especially, a Web service is available if it runs as expected in the presence of faults or stressful environments. This corresponds to the robustness definition *IEEE Standard glossary of software engineering terminology* (1999). So, it is manifest that availability implies robustness. As a consequence, the Web Service robustness must be taken into consideration in availability tests. We studied the Web Service operation robustness in Salva & Rabhi (2010): we concluded that the only robustness tests, which can be applied in SOAP environments without being blocked by SOAP processors, are requests composed of "unusual values" having a type satisfying the Web Service WSDL description. The terms "unusual values" stand for a fault in software testing Kropp et al. (1998), which gathers specific values well-known for relieving bugs. We also defined the operation robustness by focusing on the SOAP responses constructed by Web Services only. The SOAP faults, added by SOAP processors and expressing an unexpected crash, are ignored. This implies that a robust Web Service operation must yield either a response as defined in the initial specification or a SOAP fault composed of the "SOAPFaultException" cause only.

Definition 4. *Let* $\mathcal{S} =< L_\mathcal{S}, l0_\mathcal{S}, V_\mathcal{S}, V0_\mathcal{S}, I_\mathcal{S}, \Lambda_\mathcal{S}, \rightarrow_\mathcal{S} >$ *be a STS specification and* $\mathcal{S}^\uparrow$ *be its augmented STS. An operation* $op \in \Lambda_{\mathcal{S}^\uparrow}$ *is robust iff for any operation request* $?opReq(p) \in \Lambda_{\mathcal{S}^\uparrow} \times I^n$, *a SOAP message different from* $!soapfault(c) \in \Lambda_{\mathcal{S}^\uparrow} \times I$ *with* $c \neq$ *"SOAPFaultException" is received.*

The first test pattern $T1$ is derived from this definition and expresses that an operation is available if this one does not crash and responds with a SOAP message after any operation request. $T1$ means that if an operation request is "done" then it is obligatory ($\mathcal{O}$) to obtain a response $OutputResponseWS$.

$T1 \longleftrightarrow \forall opReq \in \Lambda^I_{\mathcal{S}^\uparrow}, \mathcal{O}(start(output\ OutputResponseWS)|\ done(input(opReq(p), TestDom := \{Spec(opReq); RV; Inj\})))$ where:

- $OutputResponseWS \longleftrightarrow OutputResponse(p) \vee OutputResponserobust(p)$ corresponds to a response set.
  $OutputResponse(p) = \{!opResp(p) \in \Lambda^{O}_{\mathcal{S}^{\uparrow}} \times I^{n}\}$.
  $OutputResponserobust(p) \longleftrightarrow !soapfault(''SOAPfaultException'')$ is the only SOAP fault which should be received according to Definition 4,
- $(opReq(p), TestDom := \{Spec(op); RV; Inj\})$ is a particular input modelling an operation request with parameter values in $Spec(opReq) \cup RV \cup Inj$.
  $Spec(opReq) = \{\theta \in D_{I_{\mathcal{S}^{\uparrow}}}, l \xrightarrow{?opReq(p)\varphi,\varrho}_{\mathcal{S}^{\uparrow}} l' \in \rightarrow_{\mathcal{S}^{\uparrow}}$ and $(l, v) \xrightarrow{?opReq(p),\theta} (l', v') \in \rightarrow_{||\mathcal{S}^{\uparrow}||}\}$ gathers all the values satisfying the execution of the action $?opReq(p)$. These values are given in the LTS semantics (valued automaton) of $\mathcal{S}^{\uparrow}$.
  RV is composed of random values and specific ones well-known for relieving bugs for each classical type. For instance, Figure 4 depicts the $RV(String)$ set, which gathers different values for the "String" type. RANDOM(8096) represents a random value of 8096 characters.
  $Inj \longleftrightarrow XMLInj \cup SQLInj$ corresponds to a value set allowing to perform both XML and SQL injections. $XMLInj$ and $SQLInj$ are specific value sets for the "String" type only. For instance, XML injections are introduced by using specific XSD keywords such as *maxOccurs*, which may represent a DoS (Denial of Service) attack attempt. More details about XML and SQL injections can be found in OWASP (2003).

```
<type id="String">
        <val value=null />
        <val value="" />
        <val value=" " />
        <val value="\$" />
        <val value="*" />
        <val value="&" />
        <val value="hello" />
        <val value=RANDOM(8096)" /></type>
```

Fig. 4. RV(String)

Availability is also ensured in condition that the response delay ought to be limited. This can be written with the test pattern $T2$.

$$T2 \longleftrightarrow \forall opReq \in \Lambda^{I}_{\mathcal{S}^{\uparrow}}, \mho^{\leq 60}(start(output\ OutputResponseWs)|done(input(opReq(p), TestDom := \{Spec(opReq); RV\})))$$

This one describes that for each operation request, it is obligatory to receive a response within a delay of 60s.

This test pattern can be implicitly tested if we take into account the notion of quiescence (no response observed after a timeout) during the testing process. Indeed, if quiescence is observed after a delay set to 60s, while an operation invocation, we can consider that $T2$ is not satisfied. So, this test pattern will be implicitly taken into account in Section 5.

4.2 Authentication test patterns

Authentication aims to establish or to guarantee the Client identity and to check that a Client with no credits has no permission. The logon process is often the first step in user authentication. We propose here two classical test patterns relating to this one. We suppose that the logon process is implemented classically by means of specific operation requests gathered in a set denoted $inputAuth \subseteq \Lambda^I$ which are called with authentication parameters (passwords, keys, etc.) and which return SOAP responses. $T3$ refers to the mandatory of returning a fail authentication result each time an authentication request is sent to a Web Service with unusual parameter values such as empty parameters. So, this test pattern covers the well-known empty password vulnerability:

$$T3 \longleftrightarrow \forall opReq \in inputAuth, \mho(start(output\ OutputResponseWS(rlfail))|done(input\ (opReq(p), TestDom := \{RV\}))) \text{ where:}$$

$OutputResponseWS(rlfail) \longleftrightarrow OutputResponse(rlfail) \vee OutputResponseFault(p)$. $OutputResponse(rlfail)$ represents an operation response where the message $rlfail$ in $D_{I_{s\uparrow}}$ suggests a failed login attempt. $rlfail$ must be extracted from the specification. $outputResponseFault(p) \Leftrightarrow soapfault(c)$ with $c \neq$ "Client" $\wedge\ c \neq$ "the endpoint reference not found" is a SOAP fault whose cause is different from "Client" and "the endpoint reference not found". The first one means the operation is called with bad parameter types while the second cause means that the operation name does not exist (Section 3.2).

The test pattern $T4$ is dedicated to the "brute force" threat. The latter aims to decrypt or to find authentication parameters by traversing the search space of possible values. A well-known countermeasure is to forbid a new connection attempt after n failed ones for the same user. With $n = 10$, the corresponding test pattern can be written with:

$$T4 \longleftrightarrow \forall opReq \in inputAuth, \mho(start(output\ OutputResponseWS(rlforbid))|(done((input\ (opReq(p), TestDom := \{RV\}); output\ outputResponseWS(rlfail))^{10}); done(input\ (opReq(p), TestDom := \{RV\})))) \text{ where:}$$

$OutputResponseWS(rlfail) \longleftrightarrow OutputResponse(rlfail) \vee OutputResponseFault(p)$ is an operation response as previously. The $rlfail$ message expresses a failed login attempt. The message $rlforbid$ indicates that any new connection attempt is forbidden. These messages must be extracted from the specification as well.

4.3 Authorization test patterns

Authorization represents the access policy and specifies the access rights to resources, usually for authenticated users. We define here, two test patterns which aim to check that a user, requesting for confidential data, is really authenticated.

The following test pattern checks that the request of confidential data with the operation set $inputRequestConf$, returns a "permission denied" message if the user is not authenticated (a fail login attempt has been made with the operation request $opReq_2 \in inputAuth$):

$T5 \longleftrightarrow \forall opReq \in inputRequestConf\ \exists op_2Req \in inputAuth, \mho(start(output\ OutputResponseWS(rfail))|(done(input\ op_2Req(p)); done(output\ op_2Resp(rlfail)); done(input\ (opReq(p), TestDom := \{Spec(opReq); RV\}))))$ where:

- $opResp_2(rlfail) \in \Lambda^O_{S\uparrow} \times D_I$ is an authorization operation response composed of the $rlfail$ message, which describes a fail login attempt,
- $OutputResponseWS(rfail) \longleftrightarrow OutputResponse(rfail) \vee OutputResponseFault(p)$ describes, as previously, an operation response where the message $rfail$ corresponds to an error message. $rfail$ must be extracted from the specification.

The last test pattern $T6$ is dedicated to the receipt of confidential data by means of XML or SQL injections. This one checks that an error message is received when a request containing an XML or SQL injection is sent:

$T6 \longleftrightarrow \forall opReq \in \Lambda^I_{S\uparrow}, \mho(start(output\ OutputResponseWS(rfail))|done(input\ (opReq(p), TestDom := \{Inj\})))$

4.4 Attack and vulnerability coverage

Figure 5 describes a non-exhaustive list of attacks and of vulnerabilities which are covered by the previous test patterns. This list is still extracted from the larger one given in OWASP (2003). This table also expresses the portion of Web Service vulnerabilities which shall be detected with the testing method.

Test pattern	Attacks	Vulnerabilities
T1,T2	Denial of service, special character injection, format string attack	Catch null pointer exception, deserialization of unstructured data, uncaught exception, format string, buffer overflow, improper data validation
T3	Format string attack, special character injection	Empty password, improper data validation
T4	Brute force attack	Brute force attack vulnerability, insufficient ID length
T5	Bypassing attacks	Privacy violation, failure to provide confidentiality for stored data
T6	XML, SQL injection	Missing SQL, XML validation, improper data validation

Fig. 5. Attack and vulnerability coverage

4.5 Test purpose translation

Test patterns represent abstract tests that can be used to test several Web services. Such test patterns cannot be used directly for testing since they are composed of abstract operation names. In order to derive and to execute concrete test cases, we shall translate these patterns into test requirements, called test purposes.

Test purposes describe the test intention which target some specification properties to test in the implementation. We assume that these ones are composed exclusively of specification properties which should be met in the implementation under test. Thereafter, we intend to

synchronize the STS specification with test purposes, so that final test cases will be composed of both specification behaviours and test pattern properties. So, test purposes must be formalized with STSs as well.

For a specification $\mathcal{S} =< L_{\mathcal{S}}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{\mathcal{S}}, \rightarrow_{\mathcal{S}} >$ we also formalize a test purpose with a deterministic and acyclic STS $tp =< L_{tp}, l0_{tp}, V_{tp}, V0_{tp}, I_{tp}, \Lambda_{tp}, \rightarrow_{tp} >$ such that:

- $V_{tp} \cap V_{\mathcal{S}} = \emptyset$ and V_{tp} also contains a string variable *TestDom* which is equal to the parameter domain provided in test patterns,
- $I_{tp} \subseteq I_{\mathcal{S}}$,
- $\Lambda_{tp} \subseteq \Lambda_{\mathcal{S}}$,
- $\rightarrow_{tp}$ is composed of transitions modelling specification properties. So, for any transition $l_j \xrightarrow{a(p),\varphi_j,\varrho_j}_{tp} l'_j$, it exists a transition $l_i \xrightarrow{a(p),\varphi_i,\varrho_i}_{\mathcal{S}} l'_i$ and a value set $(x_1, ..., x_n) \in D_{V \cup I}$ such that $\varphi_j \wedge \varphi_i(x_1, ..., x_n) \models$ true.

We denote TP the test purpose set derived from test patterns. In particular, a test pattern T is translated into the test purpose set $TP_T \subseteq TP$ with the following steps:

1. T is initially transformed into an abstract test purpose Atp_T, modelled with an STS, composed of generic operation requests. For a test pattern T, we denote OP_T the operation set targeted by the tests in T. For instance $OP_{T1} = \Lambda^I_{\mathcal{S}\uparrow}$,
2. the test purpose set $TP_T = \{tp_T(op) \mid op \in OP_T\}$ is then constructed by replacing generic operation invocations in Atp_T by a real operation name $op \in OP_T$.

For instance, test purpose patterns extracted from the test patterns $T1$ and $T4$ are given in Figures 6 and 7. These STSs formulate the test intention described in $T1$ and in $T4$. $T4$ describes a countermeasure for the brute force threat which is well described in the second test pattern since after ten connection attempts done by the same user, the latter cannot login anymore. *getsender* and *count* are internal procedures which return the IP address of the client and the number of times the client has attempted to connect. From the specification depicted in Figure 2, we also have $TP_{T1} = \{tp_{T1}(ItemSearchReq), tp_{T1}(ItemLookUpReq)\}$. $tp_{T1}(ItemSearchReq)$ is illustrated in Figure 8. It represents a test purpose constructed from $T1$ with the operation "ItemSearch". It illustrates the semantics of $T1$ with a concrete operation name.

Unfortunately, there is no available tool for transforming a Nomad expression into an automaton yet. At the moment, abstract test purposes must be constructed manually.

5. Testing methodology

Now, that Web services, SOAP, and security test patterns expressing security rules, are formalized, we are ready to express clearly the security level of an implementation (relative to its specification and a given set of test patterns). We initially assume that the implementation should behave like its model and can be experimented by means of the same actions. It is represented by an LTS $Impl$ and $\Delta(Impl)$ its LTS suspension. The experimentation of the implementation is performed by means of test cases defined with STSs as the specification. Test cases are defined as:

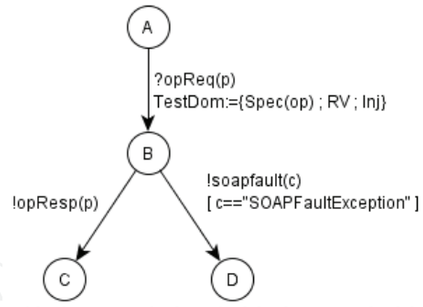


Fig. 6. Test purpose pattern derived from $T1$

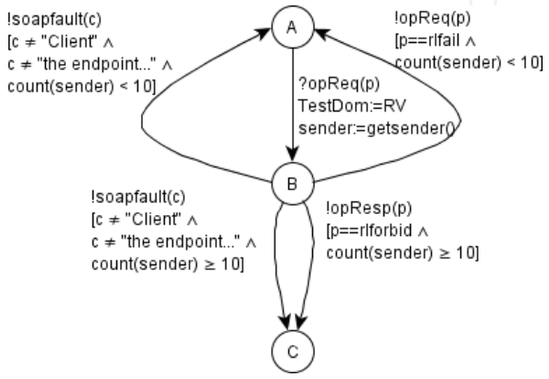


Fig. 7. Test purpose pattern derived from $T4$

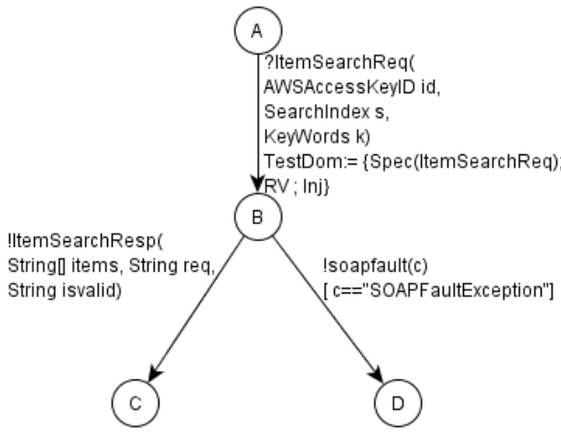


Fig. 8. A test purpose derived from $T1$

Definition 5. *A test case is a deterministic and acyclic STS* $\mathcal{TC} =< L_{\mathcal{TC}}, l0_{\mathcal{TC}}, V_{\mathcal{TC}}, V0_{\mathcal{TC}}, I_{\mathcal{TC}}, \Lambda_{\mathcal{TC}}, \rightarrow_{\mathcal{TC}} >$ *where the final locations are labelled in* $\{pass, fail\}$.

Intuitively, when the test case is executed, *pass* means that it has been completely executed, while *fail* means that the implementation has rejected it.

The proposed testing method constructs test cases to check whether the implementation behaviours satisfy a given set of security test patterns. This can be defined by means

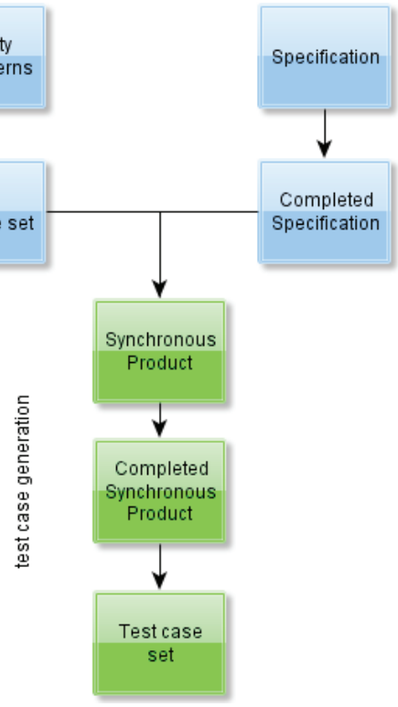


Fig. 9. Test case generation

of a relation based on traces, i.e., the observed valued actions suites expressing concrete behaviours.

More precisely, Since the implementation is seen as a black box, the method checks that the suspensions traces (actions suites) of the implementation can be found in the suspension traces of the combination of the specification with test purposes modelling concrete test patterns. We consider suspension traces, and not only traces, to take into account quiescence, i.e., lack of observation and so response delays. This can be written more formally by means of the following test relation:

$$Impl\ secure_{TP}\ \mathcal{S} \Leftrightarrow \forall tp \in TP, STraces(Impl) \cap NC\_Traces(\mathcal{S}^{\uparrow} \times tp) = \emptyset$$

with TP the test purpose set extracted from the security test patterns, $\mathcal{S}$ the specification, $\mathcal{S}^{\uparrow}$ its suspension and $NC\_Traces(\mathcal{S}^{\uparrow} \times tp) = STraces(\mathcal{S}^{\uparrow} \times tp).\Lambda^0 \cup \{!\delta\} \setminus STraces(\mathcal{S}^{\uparrow} \times tp)$ the non-conformant traces of the synchronous product $\mathcal{S}^{\uparrow} \times tp$.

To check this relation, the test case generation is performed by several steps, summarized in Figure 9 and given below. The main advantage of our model based approach, is that these steps can be automated in a tool.

1. Security test patterns are firstly translated into test purposes modelled by STSs as described in Section 4.5. For a test pattern T, we obtain a test purpose set $TP_T = \{tp_T(op) \mid op \in OP_T\}$ composed of test purposes $tp_T(op)$ with op the tested operation,
2. The specification $\mathcal{S}$ is augmented to take into consideration the SOAP environment, as described in Section 3.2,

3. The augmented specification $\mathcal{S}^{\uparrow}$ and the test purpose set TP_T are combined together: each test purpose $tp_T(op)$ is synchronized with the specification to produce the product $\mathcal{P}_T(op)$ whose paths are complete specification ones combined with the test purpose properties. We denote $Prod_T = \{\mathcal{P}_T(op) = \mathcal{S}^{\uparrow} \times tp_T(op) \mid tp_T(op) \in TP_T\}$ the resulting synchronous product set,
4. The synchronous product locations are labelled by "pass" which means that to reach this location, a correct behaviour has to be executed,
5. Synchronous products are completed on the output action set to express both correct and incorrect behaviours. A completed synchronous product is composed of *Pass* locations to express behaviours satisfying test purposes and *Fail* locations to express that test purposes and thus security test patterns are not satisfied. It results that $Prod_T^{compl} = \{\mathcal{P}_T^{compl}(op) \mid \mathcal{P}_T(op) \in Prod_T\}$ is the completed synchronous product set,
6. Finally, test cases are selected from the completed synchronous products in $Prod_T^{compl}$ by means of a reachability analysis. For a completed synchronous product $\mathcal{P}_T^{compl}(op)$, test cases in $TC_T(op)$ are STS trees which begin from the initial location of $\mathcal{P}_T^{compl}(op)$ and which aim to call the operation op. The reachability analysis ensures that these STSs can be executed on the implementation. For the test pattern T, the test case set $TC_T = \bigcup_{\mathcal{P}_T^{compl}(op) \in Prod_T^{compl}} TC_T(op)$. The final test case set TC is the union of the test case sets TC_T obtained from each test pattern T.

Each of these steps is detailed below. We assume having an augmented specification $\mathcal{S}^{\uparrow}$ and a test purpose $tp_T(op) \in TP_T$ derived from a test pattern T given in Section 4.

5.1 Synchronous product definition

A test purpose represents a test requirement which should be met in the implementation. To test this statement, both the specification and the test purpose are synchronized to produce paths which model test purpose runs with respect to the specification.

Let $tp_T(op) =< L_{tp}, l0_{tp}, V_{tp}, V0_{tp}, I_{tp}, \Lambda_{tp}, \rightarrow_{tp} >$ and $\mathcal{S}^{\uparrow} =< L_{\mathcal{S}^{\uparrow}}, l0_{\mathcal{S}^{\uparrow}}, V_{\mathcal{S}^{\uparrow}}, V0_{\mathcal{S}^{\uparrow}}, I_{\mathcal{S}^{\uparrow}}, \Lambda_{\mathcal{S}^{\uparrow}}, \rightarrow_{\mathcal{S}^{\uparrow}} >$ be two STSs. The synchronous product of $\mathcal{S}^{\uparrow}$ with $tp_T(op)$ is defined by an STS $\mathcal{P}_T(op) = \mathcal{S}^{\uparrow} \times tp_T(op) =_{def} < L_{\mathcal{P}}, l0_{\mathcal{P}}, V_{\mathcal{P}}, V0_{\mathcal{P}}, I_{\mathcal{P}}, \Lambda_{\mathcal{P}}, \rightarrow_{\mathcal{P}} >$, where:

- $L_{\mathcal{P}} = L_{\mathcal{S}} \times L_{tp}, l0_{\mathcal{P}} = l0_{\mathcal{S}} \times l0_{tp}$,
- $V_{\mathcal{P}} = V_{\mathcal{S}} \cup V_{tp}, V0_{\mathcal{P}} = V0_{\mathcal{S}} \wedge V0_{tp}$,
- $I_{\mathcal{P}} = I_{\mathcal{S}}$,
- $\Lambda_{\mathcal{P}} = \Lambda_{\mathcal{S}}$,
- $\rightarrow_{\mathcal{P}}$ is defined with the two following rules, applied successively:

$$sync : \frac{l_1 \xrightarrow{a(p),\varphi,\varrho}_{\mathcal{S}^{\uparrow}} l_2, l_1' \xrightarrow{a(p),\varphi',\varrho'}_{tp} l_2'}{(l_1 l_1') \xrightarrow{a(p),\varphi\wedge\varphi',\varrho''=[\varrho;\varrho']}_{\mathcal{P}} (l_2 l_2')}$$

$$assemble : \frac{(l_i l_j) \xrightarrow{a(p),\varphi,\varrho}_{\mathcal{P}} (l_{i+1} l_{j+1}), l_i \neq l0_{\mathcal{S}^{\uparrow}}, (l0_{\mathcal{S}^{\uparrow}} l0_{tp}) \nrightarrow (l_i l_j), l0_{\mathcal{S}^{\uparrow}} \xrightarrow{a_0(p),\varphi_0,\varrho_0} l_1 ... l_{i-1} \xrightarrow{a_{i-1}(p),\varphi_{i-1},\varrho_{i-1}} l_i \in \rightarrow_{\mathcal{S}^{\uparrow}}}{(l0_{\mathcal{S}^{\uparrow}} l0_{tp}) \xrightarrow{a_0(p),\varphi_0,\varrho_0}_{\mathcal{P}} (l_1 l_j) ... (l_{i-1} l_j) \xrightarrow{a_{i-1}(p),\varphi_{i-1},\varrho_{i-1}}_{\mathcal{P}} (l_i l_j)}$$

The first rule combines one specification transition with one test purpose one by synchronizing actions, variables updates and guards. This yields a initial transition set which is completed with the second rule to ensure that there is a specification path such that any synchronized transitions is reachable from the initial location. For sake of readability, we have denoted in the second rule $(l0_{\mathcal{S}\uparrow} l0_{tp}) \nrightarrow (l_i l_j)$ to express that there is no path from the initial location to $(l_i l_j)$ in $\mathcal{P}_T(op)$.

The synchronous product of the test purpose $tp_{T1}(ItemSearchReq)$ given in Figure 8 with the completed specification is depicted in Figure 10. The synchronized transitions obtained from the first rule are depicted in red. Initially, the test purpose aims to test the ItemSearch operation. So, the synchronous product is composed by the two ItemSearch invocations of the specification combined with test purpose properties.

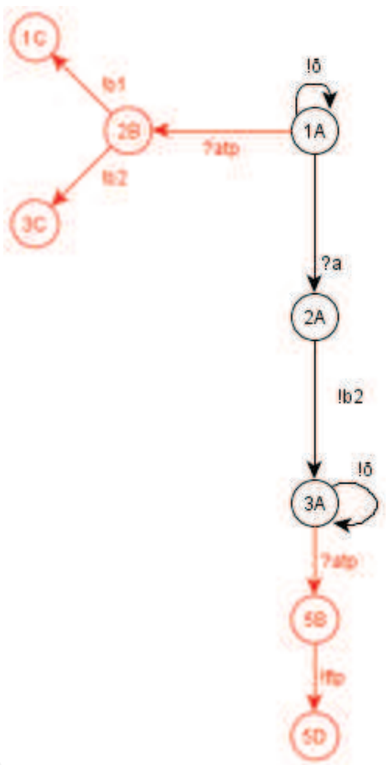


Fig. 10. A synchronous example

5.2 Incorrect behaviour completion

This straightforward part aims to complete synchronous products to express incorrect behaviours. Thanks to this steps, the generated test cases will be composed of final locations labelled either by local verdicts "pass" or "fail". The final test verdict shall be obtained without ambiguity from these local ones.

This completion is made by means of the STS operation *compl* which is defined as follows. For an STS $\mathcal{S}$, $compl\ \mathcal{S} =_{def} \mathcal{S}^{compl} =< L_{\mathcal{S}} \cup \{Fail\}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{\mathcal{S}}, \rightarrow_{\mathcal{S}^{compl}} >$ where $\rightarrow_{\mathcal{S}^{compl}}$ is obtained with the following rule:

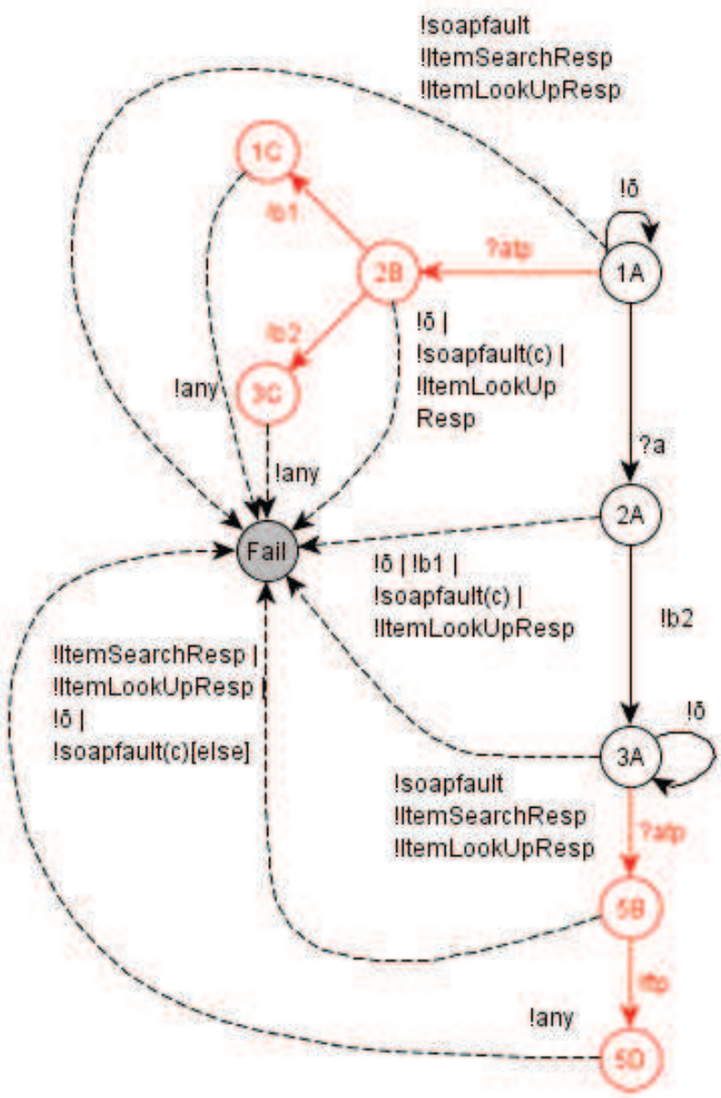


Fig. 11. A completed synchronous example

$$\frac{a \in \Lambda_{\mathcal{S}}^{O} \cup \{!\delta\},\ \varphi_a = \bigwedge_{l_1 \xrightarrow{a(p),\varphi_n,o_n}_{\mathcal{S}} l_n} \neg \varphi_n}{l_1 \xrightarrow{!a(p),\varphi_a,\emptyset}_{\mathcal{S}^{compl}} Fail}$$

A location l_1 is completed with new transitions to Fail, labelled by unexpected outputs with negations of the guards of transitions in $\mathcal{S}$.

By applying this step on the synchronous product example $\mathcal{P}_{T1}(ItemSearch)$ of Figure 10, we obtain the completed STS depicted in Figure 11. Dashed transitions depict the completion. For sake of readability, we use the label !any to model any output action. Intuitively, the dashed transitions represent unexpected output actions which lead to the Fail location. For instance, the transition $2B \xrightarrow{!\delta} Fail$ expresses that quiescence must not be observed. This transition

can be used to test the satisfaction of the test pattern $T2$ (Section 4) directly: if no response is observed after a timeout, we consider that the Web Service under test is not available and therefore faulty.

5.3 Synchronous product path extraction with reachability analysis

Test cases are extracted from the completed synchronous products with Algorithm 1. For a synchronous product $\mathcal{P}_T^{compl}(op)$, the resulting STSs in $TC_T(op)$ are trees which aim to call the operation op, referred in $tp_T(op)$ by extracting acyclic paths of $\mathcal{P}_T^{compl}(op)$ beginning from its initial location and composed of the input action $?opReq(p)$. A reachability analysis is performed on the fly to ensure that these paths can be completely executed.

The algorithm constructs a preamble by using a Depth First Path Search (DFS) algorithm between the initial location l_0 and l_k. A reachability analysis is also performed to check whether the transition t labelled by $?opReq(p)$ is reachable (lines 2-8). In line 9, the value set $Spec(opReq)$, composed of values satisfying the firing of the transition t is generated with the Solving procedure. The Value set $Value(opReq)$, composed of values used for testing op is also constructed according to the $TestDom$ variable provided in test patterns. This set may be composed of values in $Spec(opReq)$, of unusual values in RV or of SQL/XML injection values in Inj (see Section 4). SQL/XML injections are only used if the variable type is equal to "String". If the variable types are complex (tabular, object, etc.), we compose them with other types to obtain the final values. We also use an heuristic to estimate and eventually to reduce test number according to the tuple number in $Value(opReq)$. Intuitively, for a constant denoted Max, if $card(Value(opReq)) > Max$, we reduce the cardinality of $Value(opReq)$ by removing one value of $RV(type(p_1))$, and one of value of $RV(type(p_2))$, and so on up to $card(Value(opReq)) \leq Max$. This part is discussed in the next Section.

The STS tc, modelling a test case, is reset, its variables are initialized with ϱ_0. The previous preamble path and the transition labelled by the operation request $?opReq$ with one value of $Value(opReq)$ are added to the transition set of tc (lines 12-15). Then, the algorithm also adds each next transition $(l_{k+1}, l_f, !a(p), \varphi_{k+1}, \varrho_{k+1})$ with the location l_f labelled by a verdict in $\{pass, fail\}$ and transitions to $Fail$ (lines 16-19). We obtain an STS tree, which describes a complete operation invocation. tc is finally added to $TC_T(op)$.

The "Solving" method takes a path $path$ and returns a variable update ϱ_0 which satisfies the complete execution of $path$. If the constraint solvers Een & Sörensson (2003); Kiezun et al. (2009) cannot compute a value set allowing to execute $path$, then "solving" returns an empty set (lines 21-28). We use the solvers in Een & Sörensson (2003) and Kiezun et al. (2009) which work as external servers that can be called by the test case generation algorithm. The solver Kiezun et al. (2009) manages "String" types, and the solver Een & Sörensson (2003) manages most of the other simple types.

Go back to our example of Figure 11 which depicts the completed synchronous product $\mathcal{P}_{T1}^{compl}(ItemSearch)$. If we suppose having $Spec(ItemSearch) = \{(''ID'', ''book'', ''potter'')\}$ and $Inj = \{''' or '1' = '1''\}$, we obtain four test cases, two per value since the operation ItemSearch can be called two times in $\mathcal{P}_{T1}^{compl}(ItemSearch)$. Figure 12 illustrates the two test cases for the SQL injection "' or '1'='1". With the second test case, the operation ItemSearch is firstly called with ("ID","book","potter") to reach the second invocation, which is tested with the value "' or '1'='1".

Algorithm 1: STS extraction from synchronous products

Testcase(STS): TC;

input : An STS $\mathcal{P}_T^{compl}(op)$
output: A test case set $TC_T(op)$

foreach *transition* $t = (l_k \xrightarrow{?opReq(p_1,...,p_n),\varphi_k,\varrho_k}_{\mathcal{P}_T^{compl}(op)} l_{k+1}$ *with* ϱ_k *composed of the assignment TestDom* := *Domain (Section 4.5)* **do**

  repeat
    $path = DFS(l_0, l_k)$;
    $\varrho_0 := Solving(path)$;
  until $\varrho_0 \neq \emptyset$;
  if $\varrho_0 == \emptyset$ **then**
    go to next transition;
  $Spec(opReq) = \{(x_1,...,x_n) \in D_{(p_1,...,p_n)} \mid (x_1,...,x_n) := Solving(path.t)\}$;
  $Value(opReq) := \{(x_1,...,x_n) \in Spec(opReq)$ if $Spec(opReq) \in Domain\} \cup$
  $\{(x_1,...,x_n) \in D_{(p_1,...,p_n)} \mid x_i \in RV(type(p_i))$ if $RV \in Domain\} \cup$
  $\{(x_1,...,x_n) \in D_{(p_1,...,p_n)} \mid ((x'_1,...,x'_n) \in Spec(opReq), x_i = x'_i$ if $type(p_i) \neq "String", x_i \in Inj$ if $type(p_i) == "String"$, if $Inj \in Domain\}$;
  foreach $(x_1,...,x_n) \in Value(opReq)$ **do**
    $STStc := \emptyset$;
    ϱ_0 is the variable initialization of tc;
    $\varphi_{tc} := [p_1 := x_1, ..., p_n := x_n]$;
    $\rightarrow_{tc} := \rightarrow_{tc} \cup path.(l_k \xrightarrow{?opReq(p_1,...,p_n),\varphi_k \cup \varphi_{tc},\varrho_k} l_{k+1})$;
    foreach *transition* $t' = l_{k+1} \xrightarrow{!a(p),\varphi,\varrho} l_{k+2}$ **do**
      $\rightarrow_{tc} := \rightarrow_{tc} \cup t'$;
    foreach *transition* $t' = l \xrightarrow{!a(p),\varphi,\varrho}$ *Fail such that l is a location of path* **do**
      $\rightarrow_{tc} := \rightarrow_{tc} \cup t'$;
    $TC_T(op) := TC_T(op) \cup tc$;

$Solving(path\ p) : \varrho$;
$p = (l_0, l_1, a_0, \varphi_0, \varrho_0)...(l_k, l_{k+1}, a_k, \varphi_k, \varrho_k)$;
$c = \varphi_0 \wedge \varphi_1(\varrho_0) \wedge ... \wedge \varphi_k(\varrho_{k-1})$;
$(x_1,...,x_n) = solver(c)$ //solving of the guard c composed of the variables $(X_1,...,X_n)$ such that $c(x_1,...,x_n)$ true;
2 **if** $(x_1,...,x_n) == \emptyset$ **then**
  $\varrho := \emptyset$
else
  $\varrho := \{X_1 := x_1, ..., X_n := x_n\}$

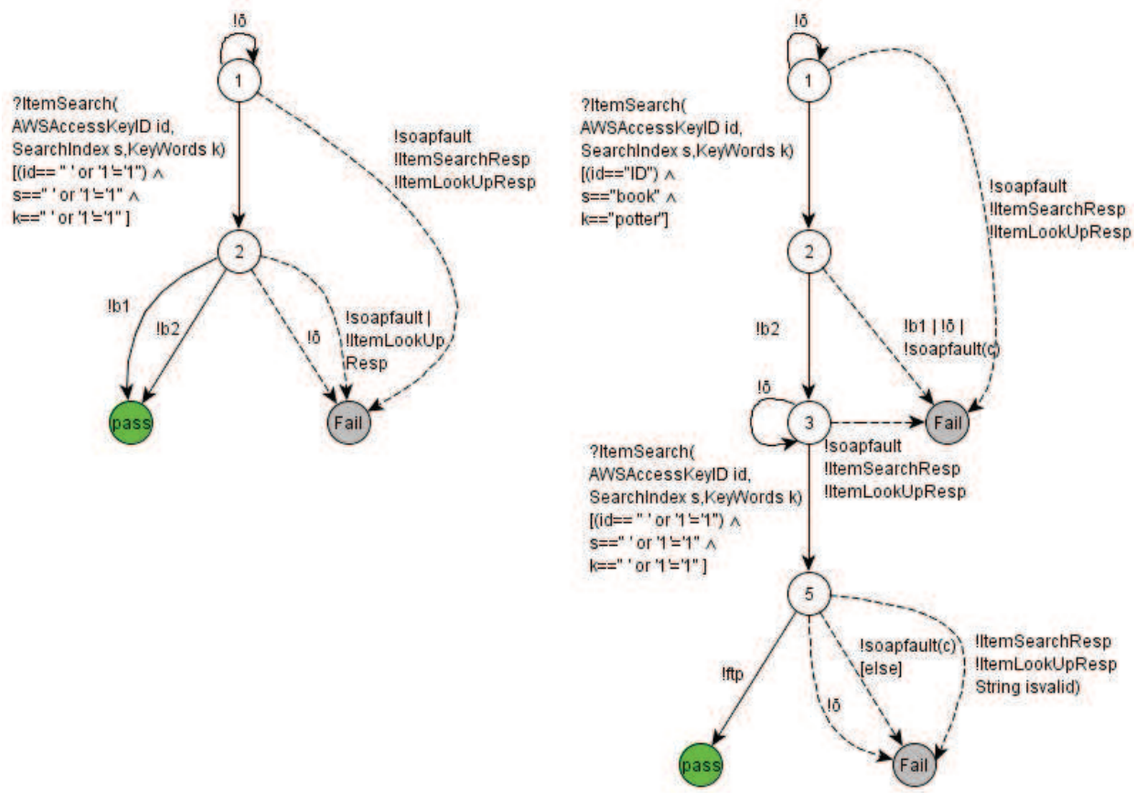


Fig. 12. Test case examples

5.4 Test verdict

In the test case generation steps, for a test purpose $tp \in TP$, we have defined the completion of the product $\mathcal{S}^{\uparrow} \times tp$ to recognizes non-conformant behaviours leading to its Fail states. So, the non-conformant trace set $NC\_STraces(\mathcal{S}^{\uparrow} \times tp)$ can be also written with the expression $STraces_{Fail}((\mathcal{S}^{\uparrow} \times tp)^{compl})$, which represents the suspension trace set leading to Fail. As a consequence, the $secure_{TP}$ relation can be also defined by:

$$\begin{aligned} Impl\ secure_{TP}\ \mathcal{S} &\Leftrightarrow \forall tp \in TP, STraces(Impl) \cap NC\_STraces(\mathcal{S}^{\uparrow} \times tp) = \emptyset \\ &\Leftrightarrow \forall tp \in TP, STraces(Impl) \cap STraces_{Fail}((\mathcal{S}^{\uparrow} \times tp)^{compl}) = \emptyset \end{aligned}$$

Now, it is manifest that the test case set, derived by our method, allows to check the satisfaction of the relation $secure_{TP}$ since a test case $\mathcal{TC} \in TC$ is selected in the product $(\mathcal{S}^{\uparrow} \times tp)^{compl}$. So, when a test case yields a suspension trace leading to a Fail state, then the implementation does not respect test purposes and security test patterns.

For a test case $\mathcal{TC}$, the suspension traces of $\mathcal{TC}$ are obtained by experimenting the implementation $Impl$. This execution of one test case $\mathcal{TC}$ on $Impl$ corresponds to the parallel composition of the LTS semantics $tc = ||\mathcal{TC}||$ with $\Delta(Impl)$, which is modelled by the LTS $\Delta(Impl)||tc =< Q_{Impl} \times Q_{tc}, q0_{Impl} \times q0_{tc}, \sum_{Impl}, \rightarrow_{\Delta(Impl)||tc} >$ where $\rightarrow_{\Delta(Impl)||tc}$ is given by the following rule:

$$\frac{q_1 \xrightarrow{a}_{\Delta(Impl)} q_2,\ q'_1 \xrightarrow{a}_{tc} q'_2}{q_1 q'_1 \xrightarrow{a}_{\Delta(Impl)||tc} q_2 q'_2}$$

Pragmatically, the tester executes a test case by covering branches of the test case tree until a *Pass* or a *Fail* location is reached. If a test case transition corresponds to an operation invocation, the latter is called with values given in the guard. Otherwise, the tester observes an event such as a response or quiescence. It searches for the next transition, which matches the observed event, and covers it.

Now, we can say that the implementation $Impl$ is $secure_{TP}$ or in other terms, satisfying a test purpose set TP, if for all test case $\mathcal{TC}$ in TC, the execution of $\mathcal{TC}$ on $Impl$ does not lead to a Fail state.

6. Experimentation and discussion

This section illustrates the benefits of using our method for security testing by giving some experiment results. We also discuss about the test coverage and the methodology complexity.

6.1 Experimentation results

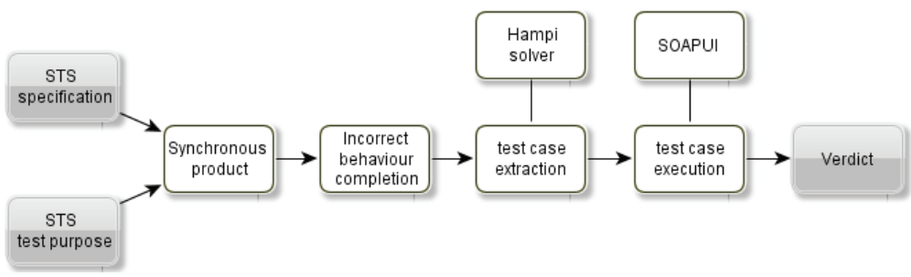


Fig. 13. Test tool architecture

We have implemented a part of this methodology in a prototype tool to experiment existing Web services. The tool architecture is illustrated in Figure 13. It performs the steps described in Section 5 i.e., synchronous products between test purposes and STS specifications, the completion of the synchronous products to add incorrect behaviours, and the test case extraction. Finally, test cases are translated into XML semi-automatically to be executed with the SOAPUI tool Eviware (2011), which is a unit testing tool for Web services. For simplicity, we have only considered String type parameters and the Hampi solver to generate values for the test case generation. To obtain a reasonable computation time, the String domain has been limited by bounding the String variable size with ten characters and by using a set of constant String values such as identification keys. We have also limited the test case number to 100. The experimentation is based upon six initial abstract test purposes, one for each test pattern given in Section 4.

Firstly, we experimented our methodology on the whole Amazon AWSECommerceService (2009/10 version) Amazon (2009). The current test purpose set had not risen security issues. Actually, this Web Service is taken as example in several research papers and many new versions of this service have been released to improve its reliability and its security. Therefore, these results are not surprising.

Web Service (WSDL)	test number	Availability	Authentication	Authorization
http://research.caspis.net/webservices/flightdetail.asmx?wsdl	56	0	0	1
http://student.labs.ii.edu.mk/ii9263/slaveProject/Service1.asmx?WSDL	60	0	1	1
http://biomoby.org/services/wsdl/www.iris.irri.org/getGermplasmByPhenotype	26	0	6	0
http://www.infored.com.sv/SRCNET/SRCWebServiceExterno/WebServSRC/servSRCWebService.asmx?WSDL	20	10	0	0
http://81.91.129.80/DialupWS/dialupVoiceService.asmx?WSDL	22	0	0	2
http://81.91.129.80/DialupWS/SecurityService.asmx?WSDL	18	0	0	3
https://intrumservice.intrum.is/vidskiptavefurservice.asmx?WSDL	66	6	1	0
http://www.handicap.fr/server_hanproducts.php?wsdl	78	2	0	4
https://gforge.inria.fr/soap/index.php?wsdl	100	1	0	0
http://193.49.35.64/ModbusXmlDa?WSDL	30	2	0	0
http://nesapp01.nesfrance.com/ws/cdiscount?wsdl	30	2	0	2
http://developer.ebay.com/webservices/latest/ShoppingService.wsdl	30	10	0	0

Fig. 14. Experimentation results

We also tested about 100 other various Web Services available on the Internet. Security vulnerabilities have been revealed for roughly 11 percent although we have a limited test purpose set. 6 percent have authorization issues and return confidential data like login, password and user-private information. Figure 14 summarizes our results.

Different kinds of issues have been collected. For instance, the Web Service *getGermplasmByPhenotype* is no more available when this one is called with special characters. Here, we suspect the existence of the "improper data validation" vulnerability. Authorization issues have been detected with *server_hanproducts.php* since its returns SOAP responses containing confidential data, such as table names and database values. Similar issues are raised with the Web Service *cdiscount*. So, these ones fail to provide confidentiality for stored data. With *slaveProject/Service1.asmx*, the "brute force" attack can be applied to extract logins and passwords.

The experimentation part has also revealed that other factors may lead to a fail verdict. For instance, the test of the *Ebay shopping* Web Service showed that quiescence was observed for a third of the operation requests. In fact, instead of receiving SOAP messages, we obtained the error "HTTP 503", meaning that the Service is not available. We may suppose here that the server was experiencing high-traffic load.

Step	Complexity	Location nb	Transition nb
Synchronous product	nn'+(n+k)n'	k	n
Completion	k	k+1	n+kn
Test case extraction	(k+1+n+kn)n×Value(opReq)	/	/

Fig. 15. Time complexity of the methodology

6.2 Discussion

Both the complexity and test coverage was left aside in the methodology description. These ones can now be discussed:

- *Methodology complexity:* the whole methodology complexity is polynomial in time in the worst case (with large test purposes testing exhaustively the implementation). This complexity is summarized in Figure 15, for one test purpose and with n (n') the specification (test purpose) transition number, k (k') the specification (test purpose) location number respectively. The *location nb* (*Transition nb*) column gives the location number (transition number) of the resulting STS once the step is achieved. In the experimentation part, we have observed that this complexity is strongly reduced since the synchronous product step produces STSs with a few more locations and transitions than the specification ones. Nevertheless, this complexity also depends on the number of testing values in $Value(opReq)$. So, if $Value(opReq)$ is large, both the complexity and the test case number may manifestly explode. This is why we implemented a heuristic which limits the test case set, by limiting the Max value in the test case extraction algorithm (Algorithm 1). When the test case number is limited to 100, testing one Web Service with our tool takes at most some minutes. The execution of 1500 tests require less than one hour. The whole test cost naturally depends on the test case number, but also on the delay required to observe quiescence. We have set arbitrarily this delay to 60s but it may be necessary to augment or to reduce it,
- *test coverage:* the test coverage of the testing method depends on the test pattern number and on the Max parameter, which represents the test number per operation. Firstly, the larger the test pattern set, the more issues that will be detected, while testing. However, our experiment results show that a non exhaustive test purpose set is already able to detect issues on a large number of Web services. The method is also scalable since the predefined set of values RV and Inj can be upgraded easily.

  The test coverage depends, besides the test pattern number, on the number of parameters per operation: the higher the number of parameters, the more difficult it will be to cover the variable space domain. This corresponds to a well-known issue in software testing. So, we have chosen a straightforward solution by bounding the test case number per operation. The Max value must be chosen according to the available time for test execution but also according to the number of parameters used with the Web service operations so that each parameter ought to be covered by a sufficient value set. For instance, for one operation composed of 4 parameters, each covered with at least 6 values, the Max parameter must be set to 1300 tests. Nevertheless, as it is illustrated in our results, a lower test case number (100 tests) is sufficient to discover security issues. There exist other interesting solutions, for the parameter coverage, which need to be investigated, such as *pairwize* testing Cohen et al. (2003) which requires that, for each pair of input parameters, every combination of values of these two parameters are covered by a test case.

7. Conclusion

We have introduced a security testing methodology dedicated for stateful Web Services. This one takes STS specifications and a Nomad test pattern set, which are translated into test purposes to check the test relation $secure_{TP}$. The specification is completed to take into account the SOAP environment while testing. Test cases are generated by means of a synchronous product between test purposes and the completed specification.

The first concluding remark, raised by our experimentation, is that SOAP Web Services are not a "security nightmare". Several companies have taken into consideration the Web Service security standards. For instance, the Amazon Service is based upon some features proposed by the WS-Security specification (timestamps, etc.). Nevertheless, our experiment results have revealed that 11 percent of the tested Web Services are vulnerable. And, we believe that this number should increase with a larger test pattern set. This leads to the first perspective. Our work is based upon the recommendations for Web Services, provided by the OWASP organization. These ones do not propose formal security rules. However, it sounds interesting to dispose, in an open-source community, of a large formal rule set, independently of the language used for modelling them. Such a rule set would be interesting to derive easily test patterns and to define the vulnerability coverage of our testing method.

Our testing tool is a prototype which requires further improvements. To the best of our knowledge, there is no Nomad parser or analyzer to translate Nomad expressions into an automata-oriented model. So, abstract test purposes are currently constructed by hands. An automatic generation would be more pleasant. The value sets, used for the test case generation can be manually modified but stay static during the test case generation. Furthermore, to avoid a test case explosion, the cardinality of these sets is reduced independently of the Web Service under test. It could be more interesting to propose a dynamic analysis of the parameter types to build a list of the most adapted values. It could be also interesting to analyze the values leading to more errors while testing and to set a weighting at each of them.

The experimentation part has also revealed that other external factors, e.g., high traffic load, may lead to a fail verdict. Such external factors show the limitations of our testing method, which cannot take them into account. A possible solution would be to complete it with a monitoring method which could detect security issues over a long period of time.

8. References

Amazon (2009). Amazon e-commerce service (ecs).

Castanet, R., Kone, O. & Laurencot, P. (1998). On the fly test generation for real time protocols, *International Conference on Computer Communications and Networks*, p. 378.

Cohen, M. B., Gibbons, P. B. & Mugridge, W. B. (2003). Constructing test suites for interaction testing, *Proc. Intl. Conf. on Software Engineering (ICSE)*, pp. 38–48.

Cuppens, F., Cuppens-Boulahia, N. & Sans, T. (2005). Nomad : A security model with non atomic actions and deadlines, *Computer Security Foundations. CSFW-18 2005. 18th IEEE Workshop*, pp. 186–196.

Darmaillacq, V., Fernandez, J., Groz, R., Mounier, L. & Richier, J.-L. (2006). Test generation for network security rules, *Testing of Communicating Systems (TestCom)*, Vol. 3964, LNCS, Springer, pp. 341–356.

Een, N. & Sörensson, N. (2003). An extensible SAT-solver, *Proc. 6th International Conference on Theory and Applications of Satisfiability Testing*, Vol. 2919, LNCS, Springer, pp. 502–518.

Eviware (2011). Soapui. http://www.soapui.org/.

Frantzen, L., Tretmans, J. & de Vries, R. (2006). Towards model-based testing of web services, *in* A. Bertolino & A. Polini (eds), *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, Palermo, Sicily, ITALY, pp. 67–82.

Frantzen, L., Tretmans, J. & Willemse, T. (2005). Test Generation Based on Symbolic Specifications, *in* J. Grabowski & B. Nielsen (eds), *Formal Approaches to Software Testing – FATES 2004*, number 3395 in *Lecture Notes in Computer Science*, Springer, pp. 1–15.

Gruschka, N. & Luttenberger, N. (2006). Protecting web services from dos attacks by soap message validation, *in Proceedings of the IFIP TC11 21 International Information Security Conference (SEC)*.

IEEE Standard glossary of software engineering terminology (1999). *IEEE Standards Software Engineering 610.12-1990. Customer and terminology standards*, IEEE press.

ir. H.M. Bijl van der, Rensink, D. A. & Tretmans, D. G. (2003). Component based testing with ioco.
URL: *http://doc.utwente.nl/41390/*

ISO/IEC (2009). Common Criteria for Information Technology Security (CC), *ISO/IEC 15408, version 3.1, ISO/IEC 15408*.

Kalam, A. A. E., Benferhat, S., Miège, A., Baida, R. E., Cuppens, F., Saurel, C., Balbiani, P., Deswarte, Y. & Trouessin, G. (2003). Organization based access control, *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, POLICY '03, IEEE Computer Society, Washington, DC, USA, pp. 120–132.
URL: *http://dl.acm.org/citation.cfm?id=826036.826869*

Kiezun, A., Ganesh, V., Guo, P. J., Hooimeijer, P. & Ernst, M. D. (2009). Hampi: a solver for string constraints, *ISSTA '09: Proc of the eighteenth international symposium on Software testing and analysis*, ACM, New York, NY, USA.

Kropp, N. P., Koopman, P. J. & Siewiorek, D. P. (1998). Automated robustness testing of off-the-shelf software components, *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, IEEE Computer Society, Washington, DC, USA, p. 230.

Le Traon, Y., Mouelhi, T. & Baudry, B. (2007). Testing security policies: going beyond functional testing, *ISSRE'07 (Int. Symposium on Software Reliability Engineering)*.
URL: *http://www.irisa.fr/triskell/publis/2007/letraon07.pdf*

Mallouli, W., Bessayah, F., Cavalli, A. & Benameur, A. (2008). Security Rules Specification and Analysis Based on Passive Testing, *in* IEEE (ed.), *The IEEE Global Communications Conference (GLOBECOM 2008)*.

Mallouli, W., Mammar, A. & Cavalli, A. R. (2009). A formal framework to integrate timed security rules within a tefsm-based system specification, *16th Asia-Pacific Software Engineering Conference (ASPEC'09), Malaysia*.

Martin, E. (2006). Automated test generation for access control policies, *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, ACM, New York, NY, USA, pp. 752–753.
URL: *http://doi.acm.org/10.1145/1176617.1176708*

Mouelhi, T., Fleurey, F., Baudry, B. & Traon, Y. (2008). A model-based framework for security policy specification, deployment and testing, *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, Springer-Verlag, Berlin, Heidelberg, pp. 537–552.

OASIS consortium (2004). Ws-security core specification 1.1. http://www.oasis- open.org/committees/tc_home.php?wg_abbrev=wss.

OASIS standards organization (2009). Xacml (extensible access control markup language). URL: *http://xml.coverpages.org/xacml.html*

OWASP (2003). Owasp testing guide v3.0 project. URL: *http://www.owasp.org/index.php/Category:OWASP_Testing _Project#OWASP_Testing_Guide_v3*

Rusu, V., Marchand, H. & Jéron, T. (2005). Automatic verification and conformance testing for validating safety properties of reactive systems, *in* J. Fitzgerald, A. Tarlecki & I. Hayes (eds), *Formal Methods 2005 (FM05)*, LNCS, Springer.

Salva, S. & Rabhi, I. (2010). Stateful web service robustness, *ICIW '10: Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services*, IEEE Computer Society, Washington, DC, USA, pp. 167–173.

Senn, D., Basin, D. A. & Caronni, G. (2005). Firewall conformance testing, *Testing of Communicating Systems (TestCom)*, Vol. 3502, LNCS, Springer, pp. 226–241.

Singh, M. & Pattterh, S. (2010). Formal specification of common criteria based access control policy, *International Journal of Network Security*, pp. 139–148.

Specification, O. U. (2002). Universal description, discovery and integration. http://www.oasisopen.org /cover/uddi.html.

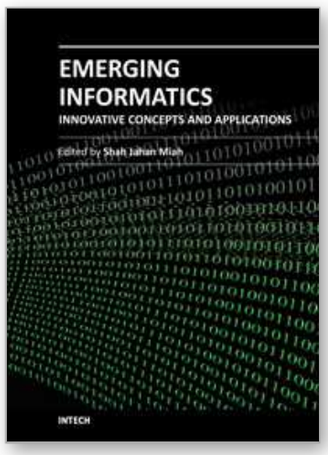
Tidwell, D. (2000). Web services, the web's next revolution, *IBM developer Works*, IBM books.

Tretmans, J. (2008). Model Based Testing with Labelled Transition Systems, *in* R. Hierons, J. Bowen & M. Harman (eds), *Formal Methods and Testing*, Vol. 4949 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, Berlin, Heidelberg, chapter 1, pp. 1–38. URL: *http://dx.doi.org/10.1007/978-3-540-78917-8_1*

World Wide Web Consortium (2001). Web services description language (wsdl).

World Wide Web consortium (2003). Simple object access protocol v1.2 (soap).

WS-I organization (2006). Basic profile. URL: *http://www.ws-i.org/docs/charters/WSBasic_ Profile _ Charter2-1.pdf, (accessed May 1, 2010)*



Emerging Informatics - Innovative Concepts and Applications

Edited by Prof. Shah Jahan Miah

ISBN 978-953-51-0514-5
Hard cover, 274 pages
Publisher InTech
Published online 20, April, 2012
Published in print edition April, 2012

The book on emerging informatics brings together the new concepts and applications that will help define and outline problem solving methods and features in designing business and human systems. It covers international aspects of information systems design in which many relevant technologies are introduced for the welfare of human and business systems. This initiative can be viewed as an emergent area of informatics that helps better conceptualise and design new world-class solutions. The book provides four flexible sections that accommodate total of fourteen chapters. The section specifies learning contexts in emerging fields. Each chapter presents a clear basis through the problem conception and its applicable technological solutions. I hope this will help further exploration of knowledge in the informatics discipline.

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen