SPaCIoS

Secure Provision and Consumption in the Internet of Services

FP7-ICT-2009-5, ICT-2009.1.4 (Trustworthy ICT)

Project No. 257876

www.spacios.eu

Deliverable D2.5.1
Framework for concretisation of abstract tests

Abstract

This deliverable discusses the development of a generic framework for model-based security testing that, firstly, captures relevant relationships between models and implementations (that is, security-specific abstractions and dual concretisations) and, secondly, operationalises the explicit knowledge of these abstractions for the semi-automated generation of the driver components. Driver components bridge the levels of abstraction between actual systems and their models, in order to execute tests.

Deliverable details

Deliverable version: *v1.0* Classification: *public*
Date of delivery: *31.03.2013* Due on: *31.03.2013*
Editors: *TUM, UNIVR, ETH Zurich, INP and IeAT principal editors; UNIGE, SAP and Siemens secondary editors* Total pages: *42*

Project details

Start date: *October 01, 2010* Duration: *36 months*
Project Coordinator: *Luca Viganò*
Partners: UNIVR, ETH Zurich, INP, KIT/TUM, UNIGE, SAP, Siemens, IeAT

SEVENTH FRAMEWORK PROGRAMME

(this page intentionally left blank)

Contents

List of Figures

List of Acronyms

AAT Abstract Attack Trace . . . 21
ASLan++ high level AVANTSSAR Specification Language . . . 6
BA Browser Action . . . 27
GA Generating Action . . . 4
MBT Model-Based Testing . . . 6
RA Recovery Action . . . 27
SUT System Under Test . . . 6
SUV System Under Validation . . . 22
TEE Test Execution Engine . . . 6
VA Verifying Action . . . 4
WAAL Web Application Abstract Language . . . 21
XSS Cross-Site Scripting . . . 30

1 Introduction

In Model-Based Testing (MBT), generating test cases from a model is only the first step. Since the model is an abstraction of the System Under Test (SUT), test cases are usually too abstract to be executed by a Test Execution Engine (TEE). Establishing relationships between models and implementations is therefore the key to be able to concretize the test cases. When models are automatically inferred from the SUT, this mapping can be defined by the extracting tool, as explained in Section 2. However, for manually written models, someone has to describe the mapping between abstract actions in the model and concrete values in the corresponding SUT. Hopefully, this manual task is simplified with the help of appropriate tools and languages.

In SPaCIoS, three driver components have been developed that support models at different level of abstraction. Thus, a modeler is free to write his model at the level of abstraction of his choice and then choose the most-suitable driver for his model. This choice mostly depends on the SUT: a protocol could be described at a low level of abstraction to simplify the concretization step, while a web application would be described at a higher level to reduce the size of the model and improve its readability.

As the abstraction level might vary from one model to another, we have to develop one driver component for each abstraction level we want to support. However, as some models in SPaCIoS are directly inferred or extracted from the SUT, we can take advantage of these model inference tools to establish some relationships between models and implementations. Gathering such information at that time is very helpful for when we need to get a concrete value of an abstract message, in order to test the corresponding implementation.

All three developed driver components are presented in Section 3: the driver for high level AVANTSSAR Specification Language (ASLan++) models at the protocol level in Section 3.1, the driver for ASLan++ models at the browser level in Section 3.2, and the driver for low-level attacker models in Section 3.3. Finally, Section 4 gives a summary of the obtained results.

2 Relationships between models and implementations

SPaCIoS uses the high-level AVANTSSAR Specification Language (ASLan++) language as a common representation for the models. While the representation is the same, the abstraction level may be different depending on the method used to create or generate the model.

2.1 Abstraction for automatically generated models

2.1.1 Inferred models

Model inference is used to generate a model of a web application automatically. The model is generated from the observed interactions between the inference algorithm and the application. As the inference works on abstract inputs and outputs, the first step of the inference process is to build a test driver which is responsible for concretisation and abstraction of input-output symbols.

A generic abstraction for web applications is defined in [9]. In this deliverable, we just provide an overview of the abstraction and concretisation provided along with the inference method. Basically, each action provided by the application through links or forms is considered as one input of the system, and each different page, from the structure point of view, is considered as one output. The collected list of inputs defines the set of abstract inputs, just as the list of pages constitute the set of abstract outputs. An abstract symbol (input or output) can be parametrized, and we just keep in the abstract form the essential parameters, whereas the concrete symbol retains all the detailed content of the real interaction with the SUT, typically a full HTTP PDU (request or response). Essential parameters are those that need to be distinguished at model level (for instance name of an agent, credential used etc.) whereas non-essential parameters include fields and details that are needed for the communication but which do not vary along a given session or do not influence the flow (e.g. content of a cookie, detailed form of a URL etc). The inference process can actually identify parameters that play a role from those that are not essential.

As explained in [9], the inference process works in two steps on web interfaces. First, in an initial crawling phase, all the inputs and outputs are collected. This phase identifies the input and output symbols, and at the same time it builds a driver to translate from concrete HTTP PDUs into abstract symbols, and conversely to concretize them. In the second phase, the inference algorithm which works at abstract level can learn a behavioral

model by interacting with the application through this driver.

As this test driver generation is done automatically, by a crawling technique, it can be used as an alternative test execution engine (TEE) or just to help the building of TEE mapping.

Model inference produces an extended finite state machine (EFSM) representation of the system. This EFSM is then converted to ASLan++, the language of the SPaCIoS tool. The translation is straightforward.

We illustrate on a Webgoat lesson the method to extract the abstract inputs from a concrete web-page. Fig 1 shows the main page of the stored XSS lesson of Webgoat and its source code (Listing 1).

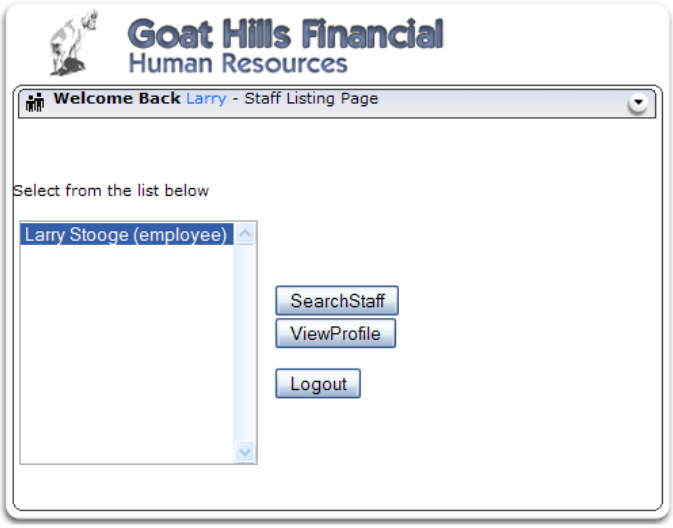

Figure 1: Stored XSS lesson main page

Listing 1: Main page of the stored XSS lesson of WebGoat

```
1  <form id="form1" name="form1" method="post" action="attack?Screen=20&menu=900">
2    <table width="60%" border="0" cellpadding="3">
3      <tr>
4        <td>
5          <label>
6            <select name="employee_id" size="11">
7              <option selected value="101">
8                Larry Stooge (employee)
9              </option>
10           </select>
11         </label>
12       </td>
13       <td>
14         <input type="submit" name="action" value="SearchStaff"/>
```

```
15          <br>
16          <input type="submit" name="action" value="ViewProfile"/>
17          <br>
18          <input type="submit" name="action" value="Logout"/>
19            </td>
20      </tr>
21    </table>
22  </form>
```

In this page, there is no link and the only inputs are identified using the *input* tags with a *submit* type. The method and the address of these inputs are extracted from the *form* tag. As the inputs share the same form element, they share the same parameters. Parameters and the possible concrete values are extracted from the *select* tag. Here we extract one parameter named *employee_id* and one possible value which is *101*. Techniques to refine the inputs, e.g, remove the unnecessary parameter from *Logout* action, are explained in [9].

We obtain the following three inputs.

```
{POST, /WebGoat/attack?Screen=20&menu=900&stage=1,
  [employee_id=101, action=Logout]},
{POST, /WebGoat/attack?Screen=20&menu=900&stage=1,
  [employee_id=101, action=ViewProfile]},
{POST, /WebGoat/attack?Screen=20&menu=900&stage=1,
  [employee_id=101, action=SearchStaff]},
```

Conversely, for concretisation, from a trace of the model, we translate each abstract input into a concrete HTTP request. Here we see that there are two levels of concretisation on the system side. The first one works at the level of HTML actions (in our example, the three actions associated with elements in the HTML output). Another concretisation is performed at the HTTP level. Concretization to HTML action level is not needed here and building the resulting HTTP requests produces the same results. An abstract input contains the method (*GET* or *POST*), the address of the page which processes the request, and the parameters. Building the corresponding HTTP request from an abstract input is straightforward since the abstraction procedure had initially collected all the values for non-essential parameters.

Let us consider the following abstract input (Method, Address, Parameters):

```
{POST, /WebGoat/attack?Screen=20&menu=900&stage=1,
[employee_id=111, password=John]}
```

The corresponding concrete HTTP request is:

```
POST /WebGoat/attack?Screen=20&menu=900&stage=1 HTTP/1.1
Host: localhost
Accept: text/html,application/xhtml+xml,application/xml;
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=
Connection: keep-alive
Cookie: JSESSIONID=49C81F65E7B54E150206769CAA7859DB
Host: localhost:8080
Referer: http://localhost:8080/WebGoat/attack?Screen=20&menu=900
User-Agent: SIMPA
Content-Length: 42
Content-Type: application/x-www-form-urlencoded

employee_id=111&password=John&Submit=Login
```

The fact that the abstraction is close to the system reduces the distance between the system and its model. In a typical scenario, an abstract attack trace (AAT) can be found. Converting this AAT to a sequence of concrete HTTP requests can be done automatically. Converting from the level of concrete actions (such as our three HTML actions in the Webgoat XSS lesson) is not needed for the inference but it is also possible to do it automatically. In addition to the relationship between a system and its model, the automatically generated test driver can be used as a test execution engine (TEE) or as a mapping provider for other TEEs.

2.1.2 Extracted models

In Deliverable 5.2 [8, Section 2.7.1] we have shown how jModex (initially named WebAppModel) extracts behavioral automata in the form of EFSMs for a system under analysis. We have also explained that, in a second phase, jModex will convert these automata into the actual ASLan++ model of the investigated program. In this section we are going to show how we plan to express in ASLan++ various information from the behavioral automata, and thus document the relations between the generated model and the system implementation.

Relationships at architectural level. For a JSP/Servlet application, jModex builds a behavioral automaton for each distinct component (i.e., servlet) of the system. All these distinct automata are translated into a single ASLan++ entity corresponding to the entire application. Consequently, the issue is how these different sub-models will be integrated into a single one. At present, the integration is performed as follows: all sub-models are enclosed into a *while(true)* loop in which one sub-model is selected for execution based on the "invoked" component. The reason behind this translation is that, in a JSP/Servlet application, the components can be invoked by a user/intruder

in any possible order without enforcing a particular invocation protocol (e.g., following the request links displayed by the presentation layer).

Relationships at code level. Each transition in a behavioral automaton produced by jModex corresponds to some execution path(s) from the analyzed program and captures the conditions under which the transition can be executed, together with all assignments to relevant state variables. In Deliverable 5.2 Figure 7 [8, Section 2.7.1] we have presented some classes of expressions (e.g., *ProgramRequest*, *SessionAttribute*, etc.) that can appear in the guards/updates of a jModex automaton. In the following we show how we plan to express in ASLan++ some of these types of expressions:

- Numeric values – ASLan++ only provides support for natural numbers. As a result, we have to abstract away the operations with real/integer values. One option is to create a particular ASLan++ type for numbers (e.g., numeric) and to treat the required numerical operations as uninterpreted functions over this type. A refinement would be to declare a distinct ASLan++ type for each numerical Java type (e.g, integer for Java int, double for Java double) in order to try to specialize the manner in which each class of values is managed during model checking.

- String values – we treat these values as ASLan++ messages. String concatenation is converted into message concatenation while the equality test is translated as message equality. It might be possible to include some special functions or predicates representing other methods on strings (e.g., equals ignoring case).

- The relevant constants identified by jModex in a program are expressed in ASLan++ as constants of corresponding types.

- Request parameter – in a JSP/Servlet application, these are simple named inputs of type string provided in the client request. We can represent them as a set of ordered pairs of messages.

- Session attribute – they represent a set of named state variables of the application and we can represent them as a set of ordered pairs of messages. Accesses/assignments to the state attributes are made using the *contains* function, while attribute deletion can be implemented using the fact retraction construct.

- Program function – not every function from the target system is analyzed in detail by jModex (e.g., library methods). Additionally, the

user may chose to ignore some methods from the program during the analysis. In these cases, the function return value is represented by jModex using a *ProgramFunction* expression which contains also the function arguments. These expressions will be converted to ASLan++ using applications of uninterpreted functions.

- Database values – as suggested by the name, these expressions represent data that come from a database. For such a value, at present, we capture the expressions corresponding to i) the SQL query producing the value and ii) to the position in the resulting table. This information can be used in ASLan++ to express database values as results obtained by applying uninterpreted functions.

- Redirection – such an expression shows that, during a transition from an automaton, the application transfers the flow of execution to another component (i.e., JSP/Servlet). This is done by telling to the user (actually, to its browser) to "invoke" another JSP/Servlet with a given set of request parameters and associated values. Consequently, this type of expression is expressed in ASLan++ as a message sending operation, from the user to the application. Additionally, a redirect expression also captures the values of the request parameters that are going to be used during the "invocation" of the new component. Usually, these values appear in a program like a constant string containing parameter-value pairs. In such cases, translating the actual request parameters of the "invocation" as ASLan++ is simple. However, the parameter-value pairs might be available just as a derived string value because, for instance, it is computed by the application based on some input value. Consequently, we cannot know precisely what request parameters are used in the redirection and which are their concrete values. In such cases, the redirection should be translated in ASLan++ as sending a message whose content might be any possible combinations of request parameters and their corresponding values.

2.2 Relation for manually built models

Models are not always extracted or inferred from the SUT. When the model is built manually, establishing the relationship between its abstract actions and the concrete values on the implementation cannot be done in an automatic way. Therefore, each driver provides tools and languages to support the modeler/tester in his task of mapping these abstract actions to concrete values that can be used by the TEE. Since the tools and/or languages

differ from one driver to another, they are described together with the driver concretization phase in the next sections.

3 Semi-automated generation of driver components

The SPaCIoS tool includes three engines for test case execution. Test case concretisation varies according to their usage. This section presents, for each engine, how drivers are generated that can execute the abstract test cases.

3.1 Test concretisation for Instrumentation-Based Testing Approach

The instrumentation-based testing approach has already been presented in Section 4.2 of Deliverable 2.1.2. We revisit it here for completeness.

The instrumentation-based testing approach executes tests in two steps. First, it instruments a model with program fragments (see Section 3.1.2). Then it executes the fragments in the order established by the attack trace (see Section 3.1.3).

3.1.1 Modeling

We specify protocols using ASLan++, whose semantics is given in ASLan. In this section we present a simplified version of ASLan, featuring only the aspects of the language that are relevant for this work. ASLan supports the specification of model checking problems of the form $M \models \phi$, where M is a labeled transition system modeling the behaviors of the honest principals and of the Dolev-Yao intruder (DY)[1] and their initial state I, and ϕ is a Linear Temporal Logic (LTL) formula stating the expected security properties.

The states of M are sets of ground (i.e. variable-free) *facts*, i.e. atomic formulae of the form given in Table 1.

Table 1: Facts and their informal meaning

Fact	Meaning
$\texttt{state}_r(j, a, [e_1, \ldots, e_p])$	a, playing role r, is ready to execute the protocol step j, and $[e_1, \ldots, e_p]$, for $p \geq 0$ is a list of expressions representing the internal state of a.
$\texttt{sent}(rs, b, a, m, c)$	rs sent message m on channel c to a pretending to be b.
$\texttt{ik}(m)$	The intruder knows message m.

[1] A Dolev-Yao intruder has complete control over the network and can generate new messages both from its initial knowledge and the messages exchanged over the network.

Transitions are represented by *rewrite rules* of the form $(L \xrightarrow{rn(v_1,\ldots,v_n)} R)$, where L and R are finite sets of facts, rn is a *rule name*, i.e. a function symbol uniquely associated with the rule, and $v_1, \ldots, v_n$ are the variables occurring in L. The variables that occur in R must also occur in L. We use typewriter font to denote states and rewrite rules with the additional convention that variables are capitalized (e.g., `Client`, `URI`), while constants and function symbols begin with a lower-case letter (e.g., `client`, `httpRequest`).

Protocol messages Messages are described as follows. HTTP requests are represented by expressions $\texttt{httpRequest}(\mathit{method}, \mathit{address}, \mathit{query\_string}, \mathit{body})$, where *method* is either the constant `get` or `post`, *address* and *query_string* are expressions representing the addressess and the query string in the URI respectively, and *body* is the HTTP body. Similarly, HTTP responses are expressions of the form $\texttt{httpResponse}(\mathit{code}, \mathit{loc}, \mathit{query\_string}, \mathit{body})$, where the *code* is either the constant `code_30x` or `code_200`, *loc* and *query_string* are (in case of redirection) the location and the query string of the location header respectively, and *body* is the HTTP body. For empty parameters, the constant `nil` is used.

Specification of the rules of the honest agents The behavior of honest principals is specified by the following rule:

$$\texttt{sent}(b_{rs}, b_i, a, m_i, c_i) \centerdot \texttt{state}_r(j, a, [e_1, \ldots, e_p]) \xrightarrow{\texttt{send}_r^{j,k}(a,\ldots)} \texttt{sent}(a, a, b_o, m_o, c_o) \centerdot \texttt{state}_r(l, a, [e'_1, \ldots, e'_q]) \quad (1)$$

for all honest principals a and suitable terms b_{rs}, b_i, b_o, c_i, c_o, $e_1, \ldots, e_p$, $e'_1, \ldots, e'_q$, m_i, m_o, and $p, q, k \in \mathbb{N}$. Rule (1) states that if principal a plays role r is at step j of the protocol and a message m_i has been sent to a on channel c_i (supposedly) by b_i, then she can send message m_o to b_o on channel c_o and change her internal state accordingly (preparing for step l). The parameter k is used to distinguish rules associated to the same principal, and role. In the initial and final rules of the protocol, the fact $\texttt{sent}(\ldots)$ is omitted in the left- and right-hand sides of the rule (1), respectively.

Specification of the intruder rules The abilities of the DY intruder of intercepting and overhearing messages are modeled by the following rules:

$$\begin{aligned} \texttt{sent}(\texttt{A}, \texttt{A}, \texttt{B}, \texttt{M}, \texttt{C}) &\xrightarrow{\texttt{intercept(A,B,M,C)}} \texttt{ik}(\texttt{M}) \\ \texttt{sent}(\texttt{A}, \texttt{A}, \texttt{B}, \texttt{M}, \texttt{C}) &\xrightarrow{\texttt{overhear(A,B,M,C)}} \texttt{ik}(\texttt{M}) \centerdot \mathit{LHS} \end{aligned} \quad (2)$$

where LHS is the set of facts occurring in the left-hand side of Rule (1).

We model the inferential capabilities of the intruder restricting our attention to those intruder knowledge derivations in which all the decomposition rules are applied before all the composition rules. The decomposition capabilities of the intruder are modeled by the following rules:

$$\mathtt{ik}(\{\mathtt{M}\}_k)\,.\,\mathtt{ik}(k^{-1}) \xrightarrow{\mathtt{decrypt(M,...)}} \mathtt{ik}(\mathtt{M})\,.\,LHS \tag{3}$$

$$\mathtt{ik}(\{\mathtt{M}\}^s_{\mathtt{K}})\,.\,\mathtt{ik}(\mathtt{K}) \xrightarrow{\mathtt{sdecrypt(K,M)}} \mathtt{ik}(\mathtt{M})\,.\,LHS \tag{4}$$

$$\mathtt{ik}(f(\mathtt{M}_1,\ldots,\mathtt{M}_n)) \xrightarrow{\mathtt{decompose}_f(\mathtt{M}_1,\ldots,\mathtt{M}_n)} \mathtt{ik}(\mathtt{M}_1)\,.\,\ldots\,.\,\mathtt{ik}(\mathtt{M}_n)\,.\,LHS \tag{5}$$

where $\{m\}_k$ (or equivalently $\mathtt{enc}(k,m)$) is the result of encrypting message m with key k and k^{-1} is the inverse key of k, $\{m\}^s_k$ (or $\mathtt{senc}(k,m)$) is the symmetric encryption, and f is a function symbol of arity $n > 0$.

For each protocol rule (1) in Section 3.1.1 and for each possible least set of messages $\{m_{1,l},\ldots,m_{j_l,l}\}$ (let m be the number of such sets, then $l = 1,\ldots,m$ and $j_l > 0$) from which the DY intruder would be able to build a message m' that unifies m_i, we add a new rule of the form

$$\mathtt{ik}(m_{1,l})\,.\,\ldots\,.\,\mathtt{ik}(m_{j_l,l})\,.\,\mathtt{state}_r(j,a,[e_1,\ldots,e_p]) \xrightarrow{\mathtt{impersonate}^{j,k,l}_r(\ldots)} \mathtt{sent}(\mathtt{i},b_i,a,m',c_i)\,.\,\mathtt{ik}(m')\,.\,LHS \tag{6}$$

This rule states that if agent a is waiting for a message m_i from b_i and the intruder is able to compose a message m' unifying m_i, then the intruder can impersonate b_i and send m'.

3.1.2 Instrumentation

The model instrumentation instructs the TEE on the generation of outgoing messages and on the checking of incoming ones. Instrumenting a model consists in calculating program fragments p associated to each rule of the model. Program fragments are then evaluated and executed by the TEE (See Section 3.1.3) in the order established by the attack trace.

Before providing further details we define how we relate expressions with actual messages. As seen in Section 3.1.1, messages in the formal model are specified abstractly. Let D be the set of data values the messages exchanged and their fields. Let E be the set of expressions used to denote data values in D. An *abstraction mapping* α maps D into E.

Let $D^{\perp}$ be an abbreviation for $D \cup \{\perp\}$ with $\perp \notin D$. Let f be a user defined function symbol of arity $n \geq 0$. Henceforth we consider constants

as functions of arity $n = 0$. We associate f to a constructor function and a family of selector functions:

Constructor: $\overline{f} : D^n \to D$ such that $\alpha(\overline{f}(d_1, \ldots, d_n)) = f(\alpha(d_1), \ldots, \alpha(d_n))$ for all $d_1, \ldots, d_n \in D$;

Selectors: $\pi^i_f : D \to D^\perp$ such that $\pi^i_f(d) = d_i$ if $d = \overline{f}(d_1, \ldots, d_n)$ and $\pi^i_f(d) = \perp$ otherwise, for $i = 1, \ldots, n$.

with the following exceptions. With $K \subseteq D$ we denote the set of cryptographic keys. If $k \in K$, then $inv(k)$ is the inverse key of k. If $f = \texttt{enc}$ (asymmetric encryption), then

1. $\pi^1_{\texttt{enc}}$ is undefined and
2. $\pi^2_{\texttt{enc}} : K \times D \to D^\perp$, written as *decrypt*, is such that $decrypt(inv(k), d') = d$ if $d' = encrypt(k, d)$ and $decrypt(inv(k), d') = \perp$ otherwise.

If $f = \texttt{senc}$, *sdecrypt* is defined similarly, replacing $inv(k)$ with k.

In security protocols specifications, the behavior of principals is represented in an abstract way, and thus the operations to check incoming messages and to generate outgoing ones are implicit. For example, in ASLan, message checks are realized by pattern matching and fields of the received message must match with some expressions stored in the state of the agent. To interact with a system under test, we need to make these procedures explicit. We write these procedures as well as the TEE in a pseudolanguage with statements such as *if-then-else*, *foreach*, and the like. We also assume that the pseudolanguage has a procedure eval(p) that evaluates a program fragment p. Let e be a ground expression in E. We denote ℓ_e a memory location that stores a data value $d \in D$ such that $e = \alpha(d)$.

A data value d could be the result of the evaluation of a program fragment p, i.e., $d = \text{eval}(p)$. We use memory locations to refer to channels as well. Let ℓ_{c_i} and ℓ_{c_o} be two memory locations for the channel constants c_i and c_o, respectively. Besides the common operation of reading and writing on channels as memory locations, we define two operators to access them as pipes in order to send (i.e. ℓ_c `>>` ℓ_m) and to receive data values (i.e. ℓ_c `<<` ℓ_m). Also, we consider a further operation to peek at the first data value available in the pipe without removing it (i.e., ℓ_c `|>` ℓ_m). The latter operator is useful to explain the instrumentation for the intruder rules.

Instrumenting honest agents The program fragment $p_{\mathtt{send}_r^{j,k}(a,\ldots,c_i,c_o)}$ encoding a rule (1) is as follows:

```
ℓ'_{m_i} := ℓ_{m_i};
ℓ_{c_i} >> ℓ_{m_i};
if ℓ'_{m_i} is not empty and ℓ_{m_i} != ℓ'_{m_i} then:
    return False;
eval(p_{m_i});
ℓ_{m_o} := eval(p_{m_o});
ℓ_{c_o} << ℓ_{m_o};
```

where m_i and m_o are the incoming and outgoing message respectively. The fragment p_{m_i} checks whether ℓ_{m_i} is such that $m_i = \alpha(\ell_{m_i})$ and p_{m_o} computes a message ℓ_{m_o} such that $m_o = \alpha(\ell_{m_o})$.

We define an association between an ASLan expression e and the fragment p used to retrieve the corresponding data value denoted by e (by accessing memory locations directly or using selectors operating on them). We call $p : e$ an *associated expression* where $e \in E$ and p is a program fragment (containing selectors operating on memory locations) such that $e = \alpha(\mathrm{eval}(p))$.

With reference to the send rule (1), just after the reception of ℓ_{m_i}, the knowledge of the principal is represented by the following set of associated expressions: $Ms = \{\ell_{m_i} : m_i, \ell_{e_1} : e_1, \ldots, \ell_{e_n} : e_n\}$. Given Ms we need compute the associated expressions of each sub-term of m_i.

Closure under decomposition Given a set Ms of associated expressions, the closure of Ms under decomposition, in symbols $\downarrow Ms$, is the smallest set such that:

1. $Ms \subseteq \downarrow Ms$,
2. if $p_1 : \mathtt{enc}(k, e) \in \downarrow Ms$ and $p_2 : \mathtt{inv}(k) \in \downarrow Ms$, then $(\mathit{decrypt}(p_2, p_1) : e) \in \downarrow Ms$,
3. if $p_1 : \mathtt{senc}(k, e) \in \downarrow Ms$ and $p_2 : k \in \downarrow Ms$, then $(\mathit{sdecrypt}(p_2, p_1) : e) \in \downarrow Ms$,
4. if $p : f(e_1, \ldots, e_n) \in \downarrow Ms$, then $(\pi_f^j(p) : e_j) \in \downarrow Ms$ for $j = 1, \ldots, n$.

For simplicity, here we assume atomic keys. Nevertheless the approach described can be readily generalized to support composed keys.

After having computed all the associated expressions, we need to either check or store the data values, according to the list of expressions representing the internal state of the principal. With reference to the send rule (1), let $kn = \{e_1, \ldots, e_n\}$, and $Ms' = \downarrow Ms - \{\ell_{e_1} : e_1, \ldots, \ell_{e_n} : e_n\}$.

Atomic checks The set of *atomic checks* P_{m_i} for a message $m_i \in E$ over a knowledge kn is defined as follows:

1. for each $p : e$ in Ms', if either e is a constant or e is a variable, and $e \in kn$ then the following fragment is in P_{m_i}:

```
if eval(p) != ℓ_e then:
    return False;
```

2. for each $p_1 : e, \ldots, p_n : e$ in Ms', if e is a variable, and $e \notin kn$ then the following fragment is a member of P_{m_i}:

```
ℓ_e := eval(p_1);
if (ℓ_e !=eval(p_2) or ℓ_e != eval(p_3) or ... or ℓ_e != eval(p_n)) then:
    return False;
```

The program fragment p_{m_i} is a sequence of all the items in P_{m_i}.

Message generation function We call *message generation function* over a set of expressions kn a function MsgGen defined as follows:

1. $\text{MsgGen}(e) = \ell_e$ if $e \in kn$;
2. $\text{MsgGen}(f(e_1, \ldots, e_n)) = \overline{f}(\text{MsgGen}(e_1), \ldots, \text{MsgGen}(e_n))$

With reference to the send rule (1), the program fragment p_{m_o} is calculated by $\text{MsgGen}(m_o)$ over $kn = \{e'_1, \ldots, e'_q\}$.

Instrumenting the intruder Let us consider the intercept rule (3) in Section 3.1.1. Let M be the message. The fragment $p_{\texttt{intercept}(A,B,M,C)}$ of pseudocode encoding the rule is as follows:

```
ℓ'_M := ℓ_M;
ℓ_c >> ℓ_M;
if ℓ'_M is not empty and ℓ_M != ℓ'_M then:
  return False;
```

where ℓ'_M contains the previous value (if any) in ℓ_M, before the reception of the new message. The fragment of pseudocode encoding the overhear rule (3) in Section 3.1.1 is the same as the one defined above, except for the operator `|>` in place of `>>`.

Let us consider the rules modeling the ability to decompose messages (i.e., `decrypt`, `sdecrypt`, and `decompose`).

The pseudocode fragment $p_{\texttt{decrypt}(M,\ldots)}$ encoding the rule (3) is as follows:

```
ℓ_M := eval(decrypt(ℓ_inv(K), ℓ_{M}_K));
```

where M and K are two ASLan expressions for the message and the public key, $\{M\}_K$ is the asymmetric encryption of M with K, and *decrypt* is the selector function associated to `enc`. Similarly for $p_{\texttt{sdecrypt}(\ldots)}$ encoding the rule (4).

The fragment $p_{\texttt{decompose}_f(M_1,\ldots,M_n)}$ encoding the rule (5) is as follows:

$$\begin{aligned}
\ell_{M_1} &:= \text{eval}(\pi_f^1(\ell_{f(M_1,\ldots,M_n)}))\,;\\
\ell_{M_2} &:= \text{eval}(\pi_f^2(\ell_{f(M_1,\ldots,M_n)}))\,;\\
&\vdots\\
\ell_{M_n} &:= \text{eval}(\pi_f^n(\ell_{f(M_1,\ldots,M_n)}))\,;
\end{aligned}$$

where $f(M_1,\ldots,M_n)$ is the message the intruder decomposes, and π_f^i for $i = 1,\ldots,n$ are the selector functions associated to the user function symbol f.

Let us consider the impersonate rule (6) in Section 3.1.1. The fragment of pseudocode $p_{\texttt{impersonate}_r^{j,k,l}(\ldots)}$ encoding this rule is computed by MsgGen(m') over the knowledge $kn = \{m_{1,l},\ldots,m_{j_l,l}\}$.

3.1.3 Test Execution Engine

The Test Execution Engine (TEE) takes as input a SUT configuration, describing which principals are part of the SUT, and an attack trace. The operations performed by the TEE are as follows:

```
1 procedure TEE(SUT : Agent Set ; [step_1, ..., step_n] : Attack Trace)
2   for i := 1 to n do :
3     if not(step_i == send_r^{j,k}(a, ...) and a ∈ SUT) then :
4       if not eval(p_{step_i}) then :
5         printf("Test execution failed in step %s", step_i);
6         halt;
```

The TEE iterates over the attack trace provided as input. During each iteration it checks whether the rule $step_i$ must be executed (line (3)). Namely, if $step_i$ is either an intruder rule or a rule concerning an agent that is not under test, then the program fragment p_{step_i} is executed. If p_{step_i} is executed without any errors the procedure continues with the next step, otherwise (lines (5)–(6)) notify that an error occurred.

3.2 Test Concretisation for SPaCiTE

SPaCiTE assumes the existence of a secure specification describing a Web application at the browser level. This specification is provided as a model

written in ASLan++. Faults are injected to this secure model by using specific mutation operators. The model-checking backends may report some Abstract Attack Traces (AATs) from such mutated models. There are then three issues in executing such AATs:

- How to instantiate an abstract action such that the TEE can execute it via a browser?
- How to instantiate the malicious parts of some abstract actions?
- How to react when the TEE fails to execute an abstract action because of missing elements?

After introducing the general framework used by SPaCiTE to instantiate an AAT at the browser level, we present solutions to the two remaining issues.

3.2.1 Framework for instantiating at the browser level

A preliminary version of this framework has been presented in [6, Section 4.3] to justify how abstract actions at the abstract testing interface can be executed by the concrete testing interface. Since this framework has been extended and its description is of essence here, we present again the parts that have not been modified, plus the extensions, in a comprehensive way.

The mapping of the abstract attack trace to executable source code is a multi-step process, as it consists of application-dependent and application-independent information. To separate the two kinds of information, we add an additional intermediate level (❷ in Figure 2) in between the abstract attack trace layer (❶ in Figure 2) and the implementation layer (❸ in Figure 2). These three layers have different purposes. Layer ❶ describes the abstract attack trace as it is given by the output of the model checker. The abstract attack trace consists of a sequence of messages that are exchanged between the defined agents. Layer ❷ describes an intermediate layer where the same abstract attack trace of layer ❶ is described using actions of Web Application Abstract Language (WAAL), a dedicated language for web applications. WAAL is a language to describe how exchanged messages between agents can be generated and verified in terms of actions a user performs in a web browser. Finally layer ❸ describes the instantiated attack trace in terms of source code. In addition it shows how the TEE reacts if an error or exception occurs during the execution of the attack trace.

First, we recall the TEE to have a clear understanding how the abstract test cases must be instantiated to be executable. Then, after describing WAAL, we present the first mapping from application-dependent messages

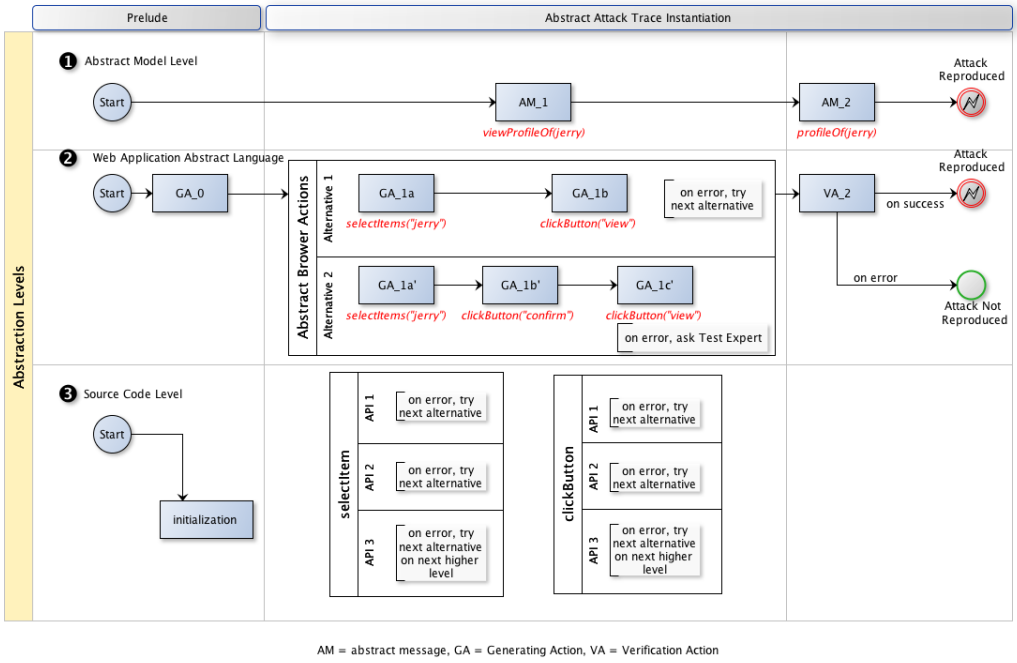

Figure 2: Layers of test case instantiation in SPaCiTE

to the intermediate level. Finally the second mapping, from the intermediate level to executable test cases, is described. As WAAL actions are application-independent, the last mapping can be reused for testing other web applications. Both mappings are illustrated with an application to the WebGoat lesson on authorization flaws.

Test Execution Engine (TEE) The TEE is responsible for running test cases and reporting verdicts. A *test case* is a sequence of descriptions of controlled and observed messages. Controlled messages are also called stimuli and observed messages are also called reactions. As such, with the reactions, a test case encodes the expected behavior with respect to the stimuli.

Running a test means *applying the stimuli* to the System Under Validation (SUV) and *observing* the SUV *reactions* (the actual reactions). Building a *verdict* means to compare the actual reactions to the expected reactions. If they conform, we say the test passes. If they do not, we say that the test fails. The result of this comparison is called the verdict. In our context, we generate *attack traces*. If we successfully reproduce an attack on a SUV, then our terminology applies as follows. As the expected reaction says that the attack should not be reproduced by the SUV (which is in conformance with

the SUV's specification), then we say that the attack has been reproduced, but the test has failed.

Having in mind this terminology, let us describe now the language used at the intermediate level.

Web Application Abstract Language (WAAL) WAAL is an abstract language for web application actions at browser level. The purpose of this language is to define actions that an end user can perform from a browser to either send messages to a Web server or check its responses. Thus, WAAL actions are split into two sets: GAs and VAs.

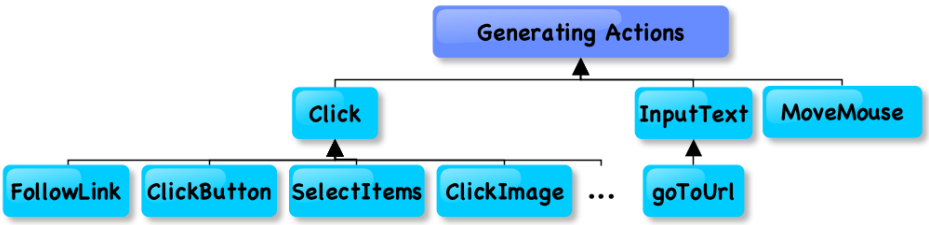

Figure 3: Generating Actions (GAs)

Generating Actions (listed in Figure 3) represent a small but complete set of atomic actions that a user can perform when he uses a web application (e.g., follow a link, click on a button, type text into a text field). More complex actions can be described by a combination of such atomic actions. For example, log in via a form may correspond to the sequence: select the name from a menu, type the password into a text field, and click on the login button. Since it works at the Browser level, GAs are close to API methods from Selenium, a Web application testing framework. However, GAs are not API methods at source code level but abstract browser actions and therefore they are technology independent.

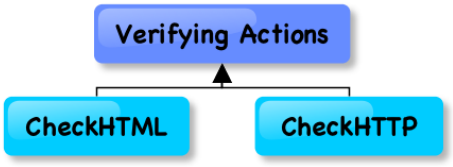

Figure 4: Verifying Actions (VAs)

Verifying Actions (listed in Figure 4) are used to verify whether an observed response matches with an expected one. A user can either verify the received message at HTML or HTTP level. The verification is performed according to a user-provided criterion.

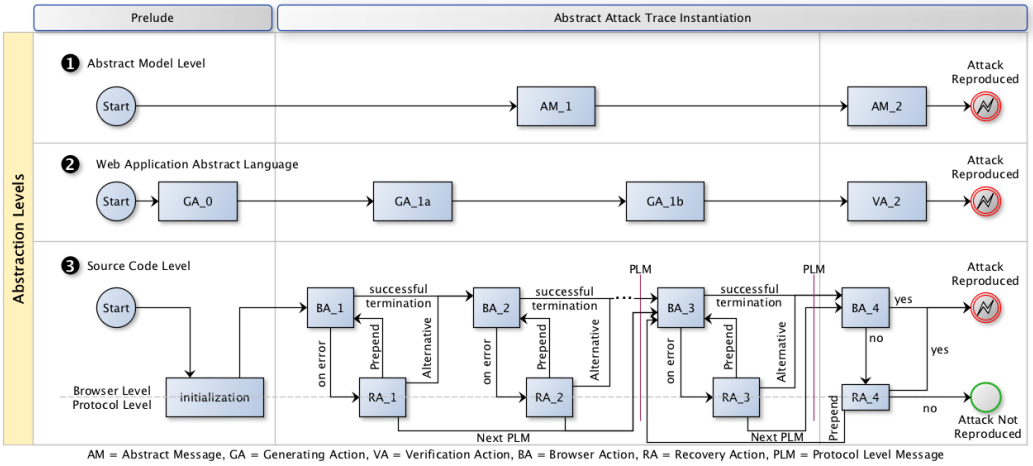

Figure 5: Instantiation and Execution Methodology

The GA and VA sets define the foundations for WAAL:

$$WAAL = (GA^* \times VA^*)^*$$

In other words, a valid word in WAAL is a sequence of actions that either produces (GA^*) or verifies (VA^*) protocol-level messages. We consider sequences only (and not trees) because this language is intended to represent the abstract attack trace at the browser level. Since a trace from a model-checker is an abstract message sequence, a sequence of actions at browser level is sufficient to represent such traces.

Mapping from abstract model level to browser level The output of the model checker is an abstract attack trace that consists of a sequence of exchanged messages. Each message m has a sender agent $\mathcal{S}$, a receiver agent $\mathcal{R}$ and a channel $\mathcal{C}$. Thus, the input layer $\mathcal{L}_1$ for the mapping to WAAL is defined as follows:

$$\mathcal{L}_1 = (A \times C \times A \times M)^*$$

where A is the set of agents, C is the type of channel used (confidential, authentic, or both), and M is the set of abstract messages exchanged between two agents.

The mapping τ_1 maps each message m (together with its sender, receiver and channel) to a pair of sequences that generates and verifies m:

$$\tau_1 : (A \times C \times A \times M) \rightarrow (GA^* \times VA^*)$$

The actual mapping depends on the sender and the receiver. Each agent described in the model is either part of the SUV — the TEE can observe his

behavior — or is simulated (stubbed) by the TEE. The former kind of agent is denoted by the set A_o, for observed agents, while the latter is denoted by the set A_s, for simulated agents. Partitioning the agent set A into A_o and A_s is the responsibility of the test expert.

Given these two sets, the sequence of GAs for $\mathcal{S} \to \mathcal{R} : m$ is constructed as follows:

$$\begin{cases} (ga_1, ga_2, \ldots, ga_n), & \text{if } \mathcal{S} \in A_s \\ (), & \text{if } \mathcal{S} \in A_o \end{cases}$$

where $n \in \mathbb{N}$ and $ga_i \in GA$ for all $1 \leq i \leq n$. Thus, if the sender $\mathcal{S}$ is a simulated agent, the message m is mapped to a sequence of GAs such that the message m is generated by a web browser after executing this sequence. If the sender is an observed agent, the TEE does not need to generate anything.

In addition to A_o and A_s, the sequence of VAs also depends on an assumption about the channel, namely whether sent messages can be assumed to be delivered unmodified. This assumption is called *integrity assumption*.

$$\begin{cases} (), & \text{if } \mathcal{S} \in A_s \wedge \text{integrity} \\ (va_1, va_2, \ldots, va_n), & \text{otherwise} \end{cases}$$

where $n \in \mathbb{N}$ and $va_i \in VA$ for all $1 \leq i \leq n$. Thus, a message m is mapped to a sequence of VAs such that a browser can verify the received message m by executing this sequence. The only case where the TEE does not need to verify m is when m has been sent over an integrity channel by a simulated agent.

In addition to mapping every message from the attack trace to sequences of actions in WAAL, the test expert must also provide an initialization block (*GA_0* in Figure 5) in order to prepare the execution of the attack trace. This initialization block is also described as actions in WAAL.

Let us now give a concrete example of this mapping, by using again the WebGoat lesson on authorization flaws.

According to the model, the agent `server` is in A_o as it is part of the SUV and the agent `tom` is in A_s as it is a compromised user and therefore must be controlled by the TEE. For our example, we also assume the integrity of messages sent over the channels. Thus, messages generated by simulated agents do not have to be verified.

Listing 2: Abstract attack trace

```
<tom>   ->*  webServer: login(tom,password(tom,server))
webServer ->   <tom> : listStaffOf(tom)
<tom>   *->  webServer: viewProfileOf(jerry)
webServer *->* <tom> : profileOf(jerry)
```

Listing 3: Mapping of WebGoat abstract actions to WAAL

```
τ1(login(usr, pwd)) =
  ((selectItem(employeeList, usr),
    intputText(passwordField, pwd),
    clickButton(login)), ())

τ1(listStaffOf(usr)) = ((),
  (checkHTML(criterionFor(listStaffOf(usr)))))

τ1(viewProfileOf(usr)) =
  ((selectItem(profileList, usr),
    clickButton(ViewProfile)), ())

τ1(profileOf(usr)) = ((),
  (checkHTML(criterionFor(profileOf(usr)))))
```

The abstract attack trace found by the model-checker (presented in Listing 2) consists of four messages (`login`, `listStaffOf`, `viewProfileOf`, `profileOf`). Listing 3 shows the mapping of these four messages to sequences of actions in WAAL; Only the GAs and VAs relevant to the attack trace are shown.

In addition to the mapping of the attack trace, the initialization block in WAAL (*GA_0*) provides a way to put the system into a state suitable to run the attack trace. For the WebGoat example, this initialization block follows some links to reach the login page of the lesson under test.

Mapping from WAAL to executable source code Once the attack trace is translated into WAAL actions, the remaining step to be able to execute the test case is to map these WAAL actions into executable statements. In contrast to the first mapping τ_1 (from abstract messages to WAAL actions) that is application dependent, the second mapping τ_2 (from WAAL to source code) is done once and for all, except if the technologies used by the TEE change.

At the source code layer, two API interfaces are used in cooperation, even though they operate on different abstraction levels. The first API works at the browser level and is then close to WAAL, which makes the translation of WAAL actions to this API easier. The second API works directly at the protocol level and is then close to Web application communication protocol. The second API is needed only if an action cannot be performed by the first API. In that case, the TEE may request the help of a test expert for

providing the corresponding protocol-level message.

In Figure 5, there are two kinds of blocks at the source code level: Browser Action (BA), and Recovery Action (RA). A BA block corresponds to an action performed on a browser. A RA block corresponds to a recovery action performed after a failure from a BA. A failure in a BA block is either a runtime exception (e.g., a browser object where an action should be invoked does not exist) or the response of the BA block corresponds to a runtime exception from the SUV (e.g. the webserver returns an "authentication required" response instead of the desired webpage). RA blocks belong to either the browser or the protocol level, depending on the failure that triggers those actions, and they may ask a test expert to provide additional information.

The mapping $\tau_2 : GA \cup VA \to (BA \times RA)^*$ maps each WAAL action to a sequence of BA and RA with $\tau_2(a) \in (BA \times RA)^*$.

If the TEE can successfully execute every BA block, which is done in a fully automatic way, then the verdict is determined as follows: if the actual reactions of the SUV conform to the expected reactions of the test cases — this verification is done by the BA blocks related to VA actions —, the attack has been reproduced and therefore the test has failed; otherwise, the test has passed.

However, a BA block may fail due to several reasons of different nature. For example, an input element is disabled, in read only mode, or its `maxLength` attribute is set to a value smaller than the size of the text to type in. Another example is a button that is disabled or totally missing, and therefore the BA block cannot click on it. For some failures of this kind, it might be possible to execute a recovering action and continue the test execution.

If an error occurs when executing a BA block, the TEE changes its operational mode and a RA block is executed in order to recover from this error (respectively, review the decision). There are three different ways of recovering after a BA block has failed: (i) prepend missing information to the BA block and execute it again; (ii) find an alternative way to execute the BA block and resume just after it; (iii) move to the protocol level, provide the corresponding message, and resume after the next protocol-level message (which may be after several BA blocks). For the following examples of these recovering methods, the browser level represents HTML elements including their actions, and the protocol level is HTTP.

As an example for the *prepend* case, let BA be an action to check the content of a webpage. This action may fail because the user is not authenticated and first has to provide credentials. In the case of basic access authentication, this request for credentials can be automatically detected and therefore it is not necessary to provide this step as WAAL actions. Thus, a possible

action in RA could be that the TEE asks the test expert for the credentials or that they are read from a configuration file. Then, the TEE reconfigures the used component by adding the credentials and re-requests the website again. Requesting a website and adding credentials can both be performed at the browser level.

For the *alternative* case, let us consider an HTML button element that triggers an event if the user clicks on it and this event is the execution of a defined JavaScript function. If the HTML button is disabled, the click event can not be triggered by the BA block. A possible alternative action, performed by the corresponding RA block, is to execute the JavaScript function directly, by using a different API call.

An example where the TEE has to switch to the protocol (HTTP) level is the following one. Assume that a BA block tries to select an element from a list and sends this value to the server by clicking on a button. This action may fail because the element is not present in the list. In that case, the TEE presents some sample HTTP messages to the test expert (e.g., by generating the HTTP messages corresponding to choosing another element from the list). Then, the TEE asks the test expert to provide the correct HTTP message. This message is sent and the BA block that follows this HTTP message is executed afterwards. It is worth noting that the underlying assumption when a RA creates some HTTP samples is that the agent state may be restored afterwards. Thus, as soon as the TEE intercepts and drops the HTTP requests, the RA block can generate as many samples as possible.

For our example, we consider the Selenium Framework at the browser level and the Apache HTTPComponents project[2] at the protocol level. Selenium provides WebElement objects to represent HTML elements and WebDriver objects to represent "the Browser" at the source code level. These objects provide API functions to find an element in the browser object (`HtmlUnitDriver.findElement()`), to access the current webpage of the browser (`HtmlUnitDriver.getPageSource()`), or to perform a click action on an HTML element (`WebElement.click()`).

Listing 4 shows how the WAAL action `selectItem`, that is part of Listing 3, is mapped to Selenium source code. The result is a Java function that takes as parameters: a connection object, the name of the list and the name of the item that should be selected from that list. The function then executes the following actions: (i) find the HTML list element, (ii) click on the corresponding item from this list. The try-catch block captures runtime exceptions and executes the corresponding RA block.

This sums up the overview of how each abstract action from a test case

[2] http://hc.apache.org/

Listing 4: Mapping of $\tau_2\,(selectItem(listID, item))$

```
Connect selectItem(Connect conn,String listID,String item){
  try {
    conn.convertToHtmlDriver();
    // Get the corresponding list
    c1 = "//a[contains(text(),'" + listID + "')]";
    list = conn.driverHtml.findElement(By.xpath(c1));
    // Select the item from the list
    c2 = "//a[contains(text(),'" + item + "')]";
    list.findElement(By.xpath(c2)).click();
  } catch (Exception e) {
    // Activate RA Block
  }
  return conn;
}
```

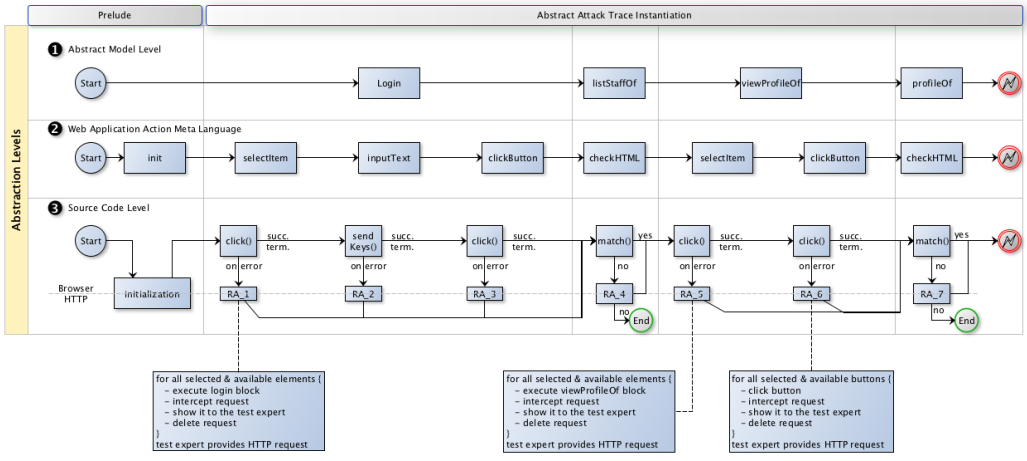

Figure 6: Instantiation

are instantiated in SPaCiTE. In the next section, we point out how malicious parameters, which are present in some abstract actions, are instantiated.

3.2.2 Getting values for malicious parameters

When the AAT comes from a mutated model, it exploits a specific injected vulnerability. Knowing which vulnerability is exploited helps in automatically executing every action of the AAT. Here, we give few examples on how this knowledge is actually used to select the malicious data used by the TEE. The complete description is available in [7, Section 1.3].

Path based access control. The abstract attack found for the mutated model of a path based access control system is reported in Listing 5. The abstract action `viewFile` corresponds, in the web application interface, to a drop-down menu with allowed filenames and a button with the label "View File" to request the selected file. The abstract action `fileContent` is designed such that the content of the requested file is displayed in a HTML page. The exact value for `notAllowed_file` and `getFileContent(notAllowed_file)` comes from an instantiation library. For example, the values `/etc/passwd` and `root:x:0:0:root:/root:/bin/bash` could be used for these two parameters, respectively.

Listing 5: Path based access control attack trace

```
user *-> <webServer>: viewFile(notAllowed_file)
<user> *-> webServer: viewFile(notAllowed_file)
webServer ->* <user>: fileContent(getFileContent(notAllowed_file))
```

Stored Cross-Site Scripting (XSS). As another example of getting values for malicious parameters, the abstract action `editProfileOf` from the attack reported in Listing 6 must inject a stored XSS payload. To get this information, we also rely on an instantiation library. For example, the payload in Listing 7 could be injected in any field of the form to edit a profile and the verification code in Listing 8 would be used when instantiating the corresponding `viewProfileOf` action.

Listing 6: Stored XSS attack trace

```
<tom>        *->*   webServer   : login(tom,password(tom,webServer))
 webServer   *->*  <tom>        : listStaffOf(tom)
<tom>        *->*   webServer   : viewProfileOf(tom)
 webServer   *->*  <tom>        : profileOf(tom)
<tom>        *->*   webServer   : editProfileOf(tom)
 webServer   *->*  <tom>        : profileOf(tom)
 jerry       *->*  <webServer>  : login(jerry,password(jerry,webServer))
```

```
<jerry>       *->*   webServer    : login(jerry,password(jerry,webServer))
 webServer    *->*  <jerry>       : listStaffOf(jerry)
<webServer>   *->*   jerry        : listStaffOf(jerry)
 jerry        *->*  <webServer>   : viewProfileOf(tom)
<jerry>       *->*   webServer    : viewProfileOf(tom)
 webServer    *->*  <jerry>       : profileOf(tom)
<webServer>   *->*   jerry        : profileOf(tom)
```

Listing 7: XSS payload for editProfileOf

```
<script>function fn(){
  var el=document.createElement('div');
  el.setAttribute('id', 'THISISANXSSATTACK');
  el.innerHTML='THISISANXSSATTACK';
  document.getElementById('lesson_wrapper').appendChild(el);}
document.getElementById('lessonContent').setAttribute('onclick', 'fn()');
</script>
```

Listing 8: XSS verification code for viewProfileOf

```
waal_click(domID("//div[@id='lessonContent']");
waal_checkInDOM(exists("//div[@id='THISISANXSSATTACK']"))
```

3.2.3 Circumventing missing elements during the execution

Intuitively, the instantiation framework goes from more abstract to more concrete in a unidirectional way. However, the TEE may fail to execute an abstract action because of missing elements in its current context. Since our TEE works at the browser level, there is an instantiation step done by the browser (i.e., from browser action to protocol-level message) that could be circumvented as well. Instead of asking the test expert to provide such missing elements, one could retrieve the missing part by analyzing the underlying relationships between abstract actions and concrete messages in another context.

This part has already been covered in details in [10, Section 4.1]. Basically, the TEE may fail to execute a translated action because it cannot reproduce it at the browser level. Still, the attack might be possible at a lower level (i.e., by bypassing the browser level). The TEE has several ways to execute the corresponding action at the HTTP level, from gathering information from previous executed actions to executing the same action in a different context (i.e., from a secure trace which is more likely to succeed). In the former, abstract parameters are bound to concrete values after the successful execution of an action; for example, the user name `jerry` is linked to the id `106` after a successful login. In the later, the abstract action is identified from the failing concrete action thanks to the relationship established by the instantiation framework. Therefore, the framework is also used in

the other direction, from concrete values to abstract actions, mostly to reuse instantiated values in a different context.

3.2.4 Requirements for the attacker behavior

One of the most important and time consuming aspects of formal validation of security specification is the intruder engine. In particular in the context of the SPaCIoS project we have used model-checkers (e.g., SATMC [2]) that use the Dolev-Yao [4] intruder for generating our test cases. While this attacker has been widely used for validating security cryptographic protocol specifications, there are indications that Dolev-Yao is not suited for web applications [1, 3]. In this section we investigate all the variations/mutations of Webgoat [5] case study, and test if we can reach the goal without encryption and decryption rule of the intruder.

We have tested 37 ASLan++ specifications and mutations of them in two different ways. The first experiment has been performed using SATMC with and without the entire intruder deduction system and the second one just excluding encryption/decryption from the set of rules of the intruder in SATMC. The first experiment has not reported the expected result for all the specifications because in one case we have reached the expected goal only using the intruder. However, we think it is important to report it in order to show that, in most of the cases, the intruder deduction system do not play any significant role to find an attack trace. On the other hand, the second experiment returns the expected result for all the specifications. That is, satmc either finds the same[3] attack with and without encryption and decryption rules, or no attack in any of two cases.

Experiment I — Excluding the deduction system of Dolev-Yao intruder. We describe here the result of testing every specification of the Webgoat case study in SPaCIoS using SATMC with and without the intruder deduction system. This experiment has been done using a beta version of SATMC 3.5 that is not publicly available yet but the developers have provided us for this specific purpose. In this beta version there is the option `--intruder_rules` that if set to `false` excludes the entire deduction system of the intruder. Unfortunately this option gives conflicts with the step compression procedure of SATMC, which we have excluded with `--sc=false`. The only side effect of this conflict is at computational level, i.e. SATMC

[3]The attack traces produced by SATMC with and without the attacker are not identical given that for the second case we don't have the intruder deduction system. But it is easy to see that the attack traces produced differ only in this aspect, so the attack traces could be considered equal from the point of view of the logic of the web application.

performances decreases, but it has not afflicted the results. Another option we had set is a threshold to decide the maximum amount of time after which a validation can be considered as inconclusive. In this experiment the time limit for inconclusive results has been (empirically) set to 300 seconds, i.e. after 300 seconds a SIGTERM signal is sent to the process.

The results of this experiment is summarized in Table 2 which reports either the SATMC output or an empty field for timeout for each specifications for the two considered cases: with and without intruder deduction system. From this table we observe that:

- for *6 specifications* we correctly find the same attack even without the attacker deduction system
- for *7 specifications* we correctly reports no attack found with and without the attacker
- *2 specifications* are reporting an attack only with the intruder
- the others *22* either do not conclude in 300 seconds (timelimit threshold) or do not report the expected attack, but it is important to highlight that the result is the same with and without the intruder

We have checked why for two specifications the attack was not found without the intruder. For `rbac_1_http_mutated.aslan++` the intruder `i` was explicitly used in the requirements. Then we have renamed it with another honest agent and the attack has been found using SATMC also without intruder.

For `xss_stored_goal2_nonSan.alan++` the attack should be also found without the intruder given that the attack trace reported by SATMC with intruder (Listing 9) does not use cryptography or pair rules.

Listing 9: xss_stored_goal2_nonSan.alan++ attack trace

```
<tom>         *->*   webServer    : login(tom,password(tom,webServer))
 webServer    *->*  <tom>         : listStaffOf(tom)
<tom>         *->*   webServer    : viewProfileOf(tom)
 webServer    *->*  <tom>         : profileOf(tom)
 jerry        *->*  <webServer>   : login(jerry,password(jerry,webServer))
<tom>         *->*   webServer    : editProfileOf(tom)
 webServer    *->*  <tom>         : profileOf(tom)
<jerry>       *->*   webServer    : login(jerry,password(jerry,webServer))
 webServer    *->*  <jerry>       : listStaffOf(jerry)
<webServer>   *->*   jerry        : listStaffOf(jerry)
 jerry        *->*  <webServer>   : viewProfileOf(tom)
<jerry>       *->*   webServer    : viewProfileOf(tom)
 webServer    *->*  <jerry>       : profileOf(tom)
<webServer>   *->*   jerry        : profileOf(tom)
```

n	Spec	With Attacker rules	Without Attacker rules
1	Command_injection_Inj	ATTACK_FOUND	ATTACK_FOUND
2	Command_injection_NoInj	NO_ATTACK_FOUND	NO_ATTACK_FOUND
3	path_based_ac_Mutated_reach	ATTACK_FOUND	ATTACK_FOUND
4	path_based_ac_Mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
5	path_based_ac	NO_ATTACK_FOUND	NO_ATTACK_FOUND
6	rbac_1_http_mutated	ATTACK_FOUND	NO_ATTACK_FOUND
7	rbac_1_http	NO_ATTACK_FOUND	NO_ATTACK_FOUND
8	rbac_1_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
9	rbac_1_no_ieqtom_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
10	rbac_1_no_ieqtom	NO_ATTACK_FOUND	NO_ATTACK_FOUND
11	rbac_1	NO_ATTACK_FOUND	NO_ATTACK_FOUND
12	rbac_3_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
13	rbac_3_no_ieqtom_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
14	rbac_3_no_ieqtom	NO_ATTACK_FOUND	NO_ATTACK_FOUND
15	rbac_3	NO_ATTACK_FOUND	NO_ATTACK_FOUND
16	stored_xss_cookie_mutated		
17	stored_xss_cookie		
18	xss_reflected_goal1_no_ieqtom	ATTACK_FOUND	ATTACK_FOUND
19	xss_reflected_goal1	ATTACK_FOUND	ATTACK_FOUND
20	xss_stored_goal1_maliciousstom		
21	xss_stored_goal1_no_ieqtom_maliciousstom		
22	xss_stored_goal1_no_ieqtom		
23	xss_stored_goal1		
24	xss_stored_goal2_maliciousTom	ATTACK_FOUND	ATTACK_FOUND
25	xss_stored_goal2_no_ieqtom_maliciousTom	ATTACK_FOUND	ATTACK_FOUND
26	xss_stored_goal2_no_ieqtom_nonsan	NO_ATTACK_FOUND	NO_ATTACK_FOUND
27	xss_stored_goal2_no_ieqtom	NO_ATTACK_FOUND	NO_ATTACK_FOUND
28	xss_stored_goal2_nonSan	ATTACK_FOUND	NO_ATTACK_FOUND
29	xss_stored_goal2	NO_ATTACK_FOUND	NO_ATTACK_FOUND
30	xss_stored_goal3_maliciousTom		
31	xss_stored_goal3_no_ieqtom_maliciousTom		
32	xss_stored_goal3_no_ieqtom		
33	xss_stored_goal3		
34	xss_stored_goal4_maliciousTom		
35	xss_stored_goal4_no_ieqtom_maliciousTom		
36	xss_stored_goal4_no_ieqtom		
37	xss_stored_goal4		

Table 2: SATMC deduction system test results

Even after adding all the needed sessions in order to let the model checker find the attack we have not managed to find this attack and it is still not clear to us if there is a bug in this beta version of SATMC with this particular set up or if there is a constraint at specification level that requires the intruder deduction system. We are investigating the core of this discrepancies. Nevertheless, this result is showing that even if we totally exclude the entire Dolev-Yao intruder deduction system and we use a standard model checker we will obtain the same result as if we were using the intruder on all the specification but one. Therefore the intruder rules seem not to be relevant to our web application case study.

Experiment II — excluding the encryption/decryption rules. The experiment has been performed using SATMC but using the standard 3.5 version instead of the beta version mentioned before. In order to exclude the encryption/decryption rules we have manually modified (deleting each occurrence of these rules) each sate[4] file generated by SATMC while running

[4]Sate files are generated by SATMC after performing a preliminary static analysis. These files contain the ASLan specifications and the intruder rules.

n	Spec	Attacker with Encryption rules	Attacker without Encryption rules
1	Command_injection_Inj	ATTACK_FOUND	ATTACK_FOUND
2	Command_injection_NoInj	NO_ATTACK_FOUND	NO_ATTACK_FOUND
3	path_based_ac_Mutated_reach	ATTACK_FOUND	ATTACK_FOUND
4	path_based_ac_Mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
5	path_based_ac	NO_ATTACK_FOUND	NO_ATTACK_FOUND
6	rbac_1_http_mutated	ATTACK_FOUND	ATTACK_FOUND
7	rbac_1_http	NO_ATTACK_FOUND	NO_ATTACK_FOUND
8	rbac_1_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
9	rbac_1_no_ieqtom_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
10	rbac_1_no_ieqtom	NO_ATTACK_FOUND	NO_ATTACK_FOUND
11	rbac_1	NO_ATTACK_FOUND	NO_ATTACK_FOUND
12	rbac_3_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
13	rbac_3_no_ieqtom_mutated	NO_ATTACK_FOUND	NO_ATTACK_FOUND
14	rbac_3_no_ieqtom	NO_ATTACK_FOUND	NO_ATTACK_FOUND
15	rbac_3	NO_ATTACK_FOUND	NO_ATTACK_FOUND
16	stored_xss_cookie_mutated		
17	stored_xss_cookie		
18	xss_reflected_goal1_no_ieqtom	ATTACK_FOUND	ATTACK_FOUND
19	xss_reflected_goal1	ATTACK_FOUND	ATTACK_FOUND
20	xss_stored_goal1_malicioustom		
21	xss_stored_goal1_no_ieqtom_malicioustom		
22	xss_stored_goal1_no_ieqtom		
23	xss_stored_goal1		
24	xss_stored_goal2_maliciousTom	ATTACK_FOUND	ATTACK_FOUND
25	xss_stored_goal2_no_ieqtom_maliciousTom	ATTACK_FOUND	ATTACK_FOUND
26	xss_stored_goal2_no_ieqtom_nonsan	NO_ATTACK_FOUND	NO_ATTACK_FOUND
27	xss_stored_goal2_no_ieqtom	NO_ATTACK_FOUND	NO_ATTACK_FOUND
28	xss_stored_goal2_nonSan	ATTACK_FOUND	ATTACK_FOUND
29	xss_stored_goal2	NO_ATTACK_FOUND	NO_ATTACK_FOUND
30	xss_stored_goal3_maliciousTom		
31	xss_stored_goal3_no_ieqtom_maliciousTom		
32	xss_stored_goal3_no_ieqtom		
33	xss_stored_goal3		
34	xss_stored_goal4_maliciousTom		
35	xss_stored_goal4_no_ieqtom_maliciousTom		
36	xss_stored_goal4_no_ieqtom		
37	xss_stored_goal4		

Table 3: SATMC Encryption test results

the standard validation on each specification of our case study. The time threshold has been set to 300 seconds as for the previous experiment.

The test is summarized in Table 3 which reports either the SATMC output or an empty field for timeout for each specifications for the two considered cases: with and without encryption rules. We can observe that for each specification the result has been the same with or without the encryption/decryption rules. In particular, we have obtained the same results as in the previous experiment except that also for `xss_stored_goal2_nonSan.alan++` SATMC with no encryption rules reports the correct attack trace.

In our experiments the Dolev-Yao intruder seems not to be relevant for web applications when it comes to cryptographic rules. We however conjecture that message concatenation, and non-deterministic scheduling of Dolev-Yao attacker would be needed on a more general web application case study. This could be used to speed up the search of goals and then to increase performances of model checkers while validating web applications.

We, therefore, want to focus on peculiarities of web applications when performing model based testing and find which is the right way to model them. This could let us use a model checker to find attacks that are not

strictly related to what the web application is implementing (e.g. protocol faults) but to the web application environment they are deep into.

As future work, we will test this conjecture by specifying a more elaborate web application, and investigate the impact of different attacker models on its verification.

3.3 Test concretisation for VERA

We have already discussed in [7] and 3.3 [10] the VERA-tool components:

- **Low-level attacker model** that reflect the common steps an attacker would take in order to exploit low-level vulnerabilities.
- **Instantiation library** containing data values used to interact with SUT.
- **Configuration file** containing system specific information needed to test a SUT.

Regarding the concretisation of abstract tests, the VERA-tool does not have a "real" concretisation phase. Since we are dealing with a low-level attacker model the concretisation phase starts during the creation of the model and it ends with the choice of the right instantiation library. In the following we show how the security expert has to take into account the missing abstraction gap between the models and the SUT, both in the creation of the models, and in the instantiation of the tests.

3.3.1 Modeling the low-level attacker

HTTP verb tempering is an attack where an attacker modifies the HTTP verb (e.g. GET, PUT, TRACE, etc.) in order to bypass access restrictions. We report the model in Figure 7; we have already explained how to create a model in Deliverable 2.4.1 so we skip the explanation on how the model works.

Every model used in the VERA-tool is created using a set of primitives defined by functions in python. We saved the used set of primitives directly in the VERA-tool source code. These primitives are the building blocks of our models; they define how to retrieve (or modify data) on the messages exchanged with the SUT.

In the scope of the VERA-tool, two approaches in the creation of the low-level attacker model primitives have emerged:

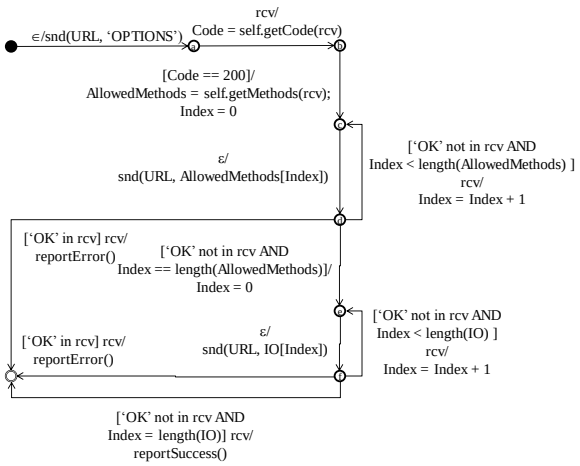

Figure 7: HTTP verb tampering low-level attacker model

- create a library of general purpose primitives in order to use it during the testing of different applications, or
- create an ad hoc library for a specific attack.

In the following we show some of the primitives (with a short explanation) for these two approaches.

General purpose primitives We currently have three general purpose modules that we use for the communication with the application, the generation of sockets, and the display of messages to the security analyst.

The **vHttp module** contains the primitives needed for HTTP communications with the applications:

Primitive	Use
snd	Sends an HTTP-Request. Request type, headers, fields, etc. can be specified. Can also automatically modify the received page to contain only absolute URLs
listForms	Returns a list of all forms on a web page
listFields	Returns a list of all fields in a form

The **vSocket module** specifies the primitives for the managing of sockets:

Primitive	Use
openSocket/listen	Creates a new socket and starts listening
connect	Connects to a remote host
snd	Sends data over the established connection
isConnected	Returns whether a connection has been established
close	Closes socket/connection

The **vPrint module** contains the primitives that we use for the display of messages to the security analyst.

Primitive	Use
debug info warning normal success fail	Returns a message of the used type to the user

Ad hoc primitives The second approach consists in the creation of primitives for targeting a specific attack. In the following we report two attacks with the primitives used in the corresponding low-level attacker model.

XML Signature wrapping attack

Primitive	Use
snd	to send HTTP/HTTPS requests
getIdP	to obtain IdP from a SAMLL Authentication Request message
getSAMLRequest	to obtain SAML Request from a SAML Authentication Request message
getRelayState	to obtain relay state from a SAML message
getBody	to create a message body to be sent to IdP
getSAMLResponse	to obtain SAML Response from a SAML Authentication Response message
fuzz	to apply XML Signature wrapping attack on SAML Response
getTarget	to obtain target URL to send the changed SAML Response
reportError	to report error in SUT
reportSuccess	to report no error in SUT
clearCookies	to clear cookies received

```
IO=[
 ["<IMG SRC%3d\"javascript:alert('XSS');\">","
   <IMG SRC=\"javascript:alert('XSS');\">"],
 ["<IMG \"\"\"><SCRIPT>alert(\"XSS\")</SCRIPT>\">","
   <IMG \"\"\"><SCRIPT>alert(\"XSS\")</SCRIPT>\">"],
 ["<script>alert(1);</script>",
  "<script>alert(1);</script>"]
]
```

(a) IO for Cross-Site Scripting.

```
IO=[
   ["'","nexpected"],
   ["\"","nexpected"],
   ["1 OR 1=", "nexpected"],
   ["123 OR 1=1--", "Congratulations"],
   ["' OR 1=1--", "Congratulations"]
]
```

(b) IO for SQL-injection

Figure 8: Example of Instantiation Libraries used in VERA

Verb tampering attack

Primitive	Use
snd	of course;
getCode	to obtain the code of HTTP response message;
getMethods	to obtain the allowed methods of the server;
reportSuccess	
reportError	

3.3.2 Prioritizing Instantiation Library

We already discussed how to prioritize the attacker models of VERA (approach based on a risk analysis of the system under test (SUT)) in Section 3.1.1 of Deliverable 3.3.

As stated in the deliverable the set V containing the available low-level attacker models $v_1, \cdots, v_n$ can be reduced to a subset V_e (the vulnerabilities to be tested on the SUT) according to a risk factor. We have show how these models use the instantiation libraries according to the vulnerability that they implement. An example of Instantiation Libraries used by the VERA-tool during the assessment of i) a general SQL-injection or ii) a Cross-Site Scripting attacks is shown in Figure 8. An Instantiation Library consists of a simple text file containing an array called IO of either single values or tuples. Which kind of data is in the array depends largely on the attacker model it was created for.

Write $\alpha(v)$ for the set of instantiation libraries associated to a vulnerability $v \in V$. In order to choose the right instantiation library during the test, we introduced the concept of *tags*; a tag is written in the form `c.v`, where `c` is a *category* and `v` is a *value*. We assume that the SUT is tagged according with the values used for tagging the instantiation libraries.

We have identified two common scenarios where it makes sense to use multiple Instantiation Libraries during a vulnerability assessment:

- If the same steps can be performed to create completely different attacks, then it makes sense to use different Instantiation Libraries for

these kinds of attacks. It makes sense to not have a single big library, but rather have different instantiation libraries $\alpha(v_1), \ldots, \alpha(v_n)$.

- If there are a large number of values, and these can be divided based on information which might be available during the test time, such as the back-ends used, it might make sense to split the Instantiation Library ($\alpha(v) = \mathrm{IO}_1 \cup \ldots \cup \mathrm{IO}_n$). An example for this would be file enumeration, where different instantiation libraries might exist for the different kinds of platforms available.

For the first case we can use as an example the two instantiation libraries shown in Figure 8. The low-level attacker model injection attacks (e.g. SQL-, X-Path-, Command-Injection, etc.) is the same, the only thing that change is the instantiation library that the security analyst has to select during the vulnerability assessment.

As an example of for the second case we report the different queries (with the corresponding tag) that a security analyst can try in order to assess the version of the database of the SUT.

bd.db2

```
select versionnumber,
       version_timestamp
        from sysibm.sysversions;
```

bd.mssql & *bd.mysql*	*bd.postgresql*
`select @@version`	`select version()`

With multiple instantiation libraries, the corresponding tags, and the tags for the SUT we can calculate the likelihood of each instantiation library in $\alpha(v)$ (refer to Section 3.1.1 of Deliverable 3.3 for the algorithm for calculating likelihood). Say that $l(L)$ is the likelihood of succeeding in performing the test on the SUT using the instantiation library $L \in \alpha(v)$. For example if the security analyst has to assess a SUT tagged with `db.mysql` the likelihood of the instantiation libraries tagged with `db.db2` and `db.postgresql` is zero.

4 Summary

This deliverable has presented how abstract test cases generated in SPaCIoS are turned into concrete test cases that can be executed by one of the TEE developed in the project.

First of all, the relationship between abstract actions in the model and concrete data in the SUT is established. This can be done in a more or less automatic way when the model is inferred or extracted from the SUT. For manually created models, the modeler can describe this mapping using some intermediate languages, developed within the project, which make this task easier and less error-prone. TEEs are able to automatically execute test cases described in such languages.

Then, concretisation of test cases is explained for each driver component. As these drivers support different models, the concretisation phase slightly differs from one to another. However, they all make use of the established relationships between models and implementations. At the end, it is possible to concretize tests for both security protocols and Web applications.

Some issues remain open at the time of writing this deliverable. These open questions will be addressed in the following deliverable on bridging components for different levels of abstraction (D3.4.1).

References

[1] D. Akhawe, A. Barth, P. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 290–304, July.

[2] A. Armando and L. Compagna. SATMC: a SAT-based model checker for security protocols. In *Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA'04)*, volume 3229 of *LNAI*, pages 730–733, Lisbon, Portugal, 2004. Springer-Verlag.

[3] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *25th IEEE Computer Security Foundations Symposium, CSF 2012*, June 2012.

[4] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. on Information Theory*, IT-29(2):198–208, 1983.

[5] OWASP. OWASP WebGoat Project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, 2011.

[6] SPaCIoS. Deliverable 2.1.2: Modeling security-relevant aspects in the IoS, 2012.

[7] SPaCIoS. Deliverable 2.4.1: Definition of Attacker Behavior Models, 2012.

[8] SPaCIoS. Deliverable 5.2: Proof of Concept and Tool Assessment v.2, 2012.

[9] SPaCIoS. Deliverable 2.2.1: Method for assessing and retrieving models, 2013.

[10] SPaCIoS. Deliverable 3.3: SPaCIoS Methodology and technology for vulnerability-driven security testing, 2013.