From Model-checking to Automated Testing of Security Protocols: Bridging the Gap[⋆]

Alessandro Armando[1,2], Giancarlo Pellegrino[3,4], Roberto Carbone[2], Alessio Merlo[1,5], and Davide Balzarotti[3]

[1] DIST, Università degli Studi di Genova, Italy
{armando, alessio.merlo}@dist.unige.it
[2] Security & Trust Unit, FBK-irst, Trento, Italy
{armando, carbone}@fbk.eu
[3] Institute Eurecom, Sophia Antipolis, France
{giancarlo.pellegrino, davide.balzarotti}@eurecom.fr
[4] SAP Research, Mougins, France
giancarlo.pellegrino@sap.com
[5] Università Telematica E-Campus, Italy
alessio.merlo@uniecampus.it

Abstract. Model checkers have been remarkably successful in finding flaws in security protocols. In this paper we present an approach to binding specifications of security protocols to actual implementations and show how it can be effectively used to automatically test implementations against putative attack traces found by the model checker. By using our approach we have been able to automatically detect and reproduce an attack witnessing an authentication flaw in the SAML-based Single Sign-On for Google Apps.

1 Introduction

Security protocols are communication protocols that aim at providing security guarantees (such as authentication or confidentiality) through the application of cryptographic primitives. Security protocols lie at the core of security-critical applications, such as Web-based Single Sign-On solutions and on-line payment systems. Unfortunately, security protocols are notoriously error-prone as witnessed by the many protocols that have been found vulnerable to serious attacks years after their publication and implementation. (See [13] for a survey.)

Interestingly, many attacks on security protocols can be carried out without breaking cryptography. These attacks exploit weaknesses in the protocols that are due to the complex and unexpected interleaving of different protocol sessions as well as to the possible interference of malicious agents. Since these weaknesses are very difficult to spot by traditional verification techniques

[⋆] This work has partially been supported by the FP7-ICT Project SPaCIoS (no. 257876) and by the project SIAM funded in the context of the FP7 EU "Team 2009 - Incoming" COFUND action.

(e.g., manual inspection and testing), a variety of novel model checking techniques specifically tailored to the analysis of security protocols have been put forward [1, 18, 22]. This has spurred the development of a new generation of model checkers which has proved remarkably successful in discovering (previously unknown) flaws in security protocols [4, 20, 28]. While in the past model checkers have been mainly used to support the analysis of security protocols at design time, recently their usage has been extended to support the discovery of vulnerabilities in actual, even deployed, systems. For instance, model checking was key to the discovery of serious vulnerabilities in the SAML-based Single Sign-On for Google Apps [3] as well as in the PKCS#11 security tokens [11]. The main limitation of the existing approaches is that reproducing attack traces found by a model checker against protocol implementations not only requires a thorough understanding of both the protocol and its implementation, but also a substantial amount of manual activity.

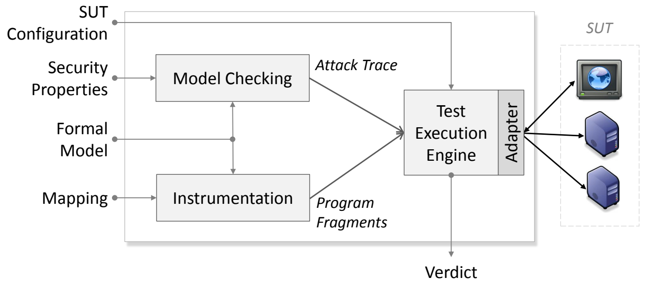


Fig. 1. Overview of the Approach

In this paper we tackle this difficulty by presenting an approach that supports *(i)* the binding of specifications of security protocols to actual implementations through model instrumentation, and *(ii)* the automatic testing of real implementations against putative attacks found by a model checker.

It is worth pointing out that most model checking techniques (and the associated tools) for security protocol analysis work on abstract models of the protocols. These models do not specify how protocol messages should be checked and generated, nor the way in which the internal state of the principals should be updated. As a consequence, the attack traces returned by these tools are not directly executable. Our paper shows that this gap can be filled in automatically. To the best of our knowledge a solution to this problem is not available.

Our approach consists of the following steps (cf. Figure 1):
Model Checking. Given a formal model of the protocol and a description of the expected security properties, a model checker systematically explores the state space of the model looking for counterexamples. Any counterexample found by the model checker is returned as an *Attack Trace*.
Instrumentation. The instrumentation step automatically calculates and provides the *Test Execution Engine* with a collection of *Program Fragments*, encoding how to verify (generate) incoming (outgoing, resp.) messages, by using the functionalities provided by the *Adapter*. The association between abstract messages and concrete ones is in the *Mapping* input.
Execution. The *Test Execution Engine* (TEE) interprets the *Attack Trace* and executes the program fragments accordingly. The *SUT Configuration* specifies

which principals are part of the System Under Test (SUT) and which, instead, are simulated by the TEE. The *Verdict* indicates whether the TEE succeeded or not in reproducing the attack. Note that if the verdict is negative, the whole approach can be iterated by requesting the model checker to provide another attack trace (if any).

Our approach naturally supports both model and property-driven security testing and in doing so it paves the way to a range of security testing techniques that go beyond those implemented in state-of-the-art penetration testing tools [9, 15]. For instance, prior research has shown that a number of subtle flaws found by model checkers can be exploited in real implementations as launching pad for severe attacks [3, 4, 11]. Moreover, even when security protocols do not suffer from design flaw, their implementations can still expose vulnerabilities. In these cases mutants can be derived from the original model [12, 14] and our approach can be used to check their existence into the implementation.

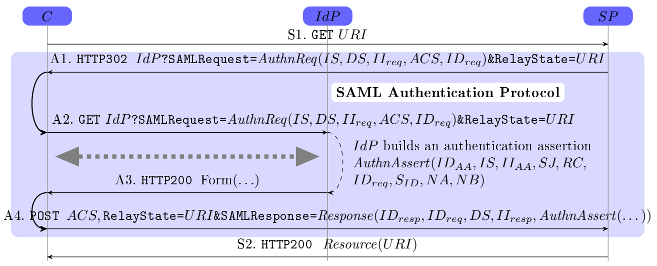
In order to assess the effectiveness of the proposed approach we developed a prototype of the architecture in Figure 1 and used it to test two Web-based Single Sign-On (SSO) solutions that are available on-line, namely the SAML-based SSO for Google Apps and the SimpleSAMLphp SSO service offered by Foodle. The prototype is able to successfully execute an attack on the Google service whereby a client gets access to her own Gmail account without having previously requested it [4]. Quite interestingly, our prototype also shows that the same attack does not succeed against the SSO service of Foodle, due to specific implementation mechanisms used by SimpleSAMLphp.

2 SAML Web-browser SSO

Browser-based Single Sign-On (SSO) is replacing conventional solutions based on multiple, domain-specific credentials by offering an improved user experience: clients perform a single log in operation to an identity provider, and are yet able to access resources offered by a variety of service providers. Moreover, by replacing multiple credentials (one per service provider) with a single one (associated with the identity provider), SSO solutions are expected to improve the overall security as users tend to use weak passwords and/or to reuse the same password on different service providers.

The OASIS *Security Assertion Markup Language* (SAML) 2.0 Web Browser SSO Profile (SAML SSO, for short) [23] is an emerging standard for Web-based SSO. Three basic roles take part in the protocol: a client C, an identity provider IdP and a service provider SP. The objective of C, typically a web browser guided by a user, is to get access to a service or a resource provided by SP. IdP is responsible to authenticate C and to issue the corresponding authentication assertions (a special type of assertion used to authenticate users). The SSO protocol terminates when SP consumes the assertions generated by IdP to grant or deny C access to the requested resource.

Figure 2 shows an excerpt of the messages exchanged during a typical execution of the SAML SSO protocol. In the first message (S1), C asks SP to provide the resource located at URI. SP then initiates the protocol by sending C a redirect response (A1) of the form:



C
IdP
SP
S1. GET URI
A1. HTTP302 IdP?SAMLRequest=$AuthnReq(IS, DS, II_{req}, ACS, ID_{req})$&RelayState=$URI$
SAML Authentication Protocol
A2. GET IdP?SAMLRequest=$AuthnReq(IS, DS, II_{req}, ACS, ID_{req})$&RelayState=$URI$
IdP builds an authentication assertion $AuthnAssert(ID_{AA}, IS, II_{AA}, SJ, RC, ID_{req}, S_{ID}, NA, NB)$
A3. HTTP200 Form(...)
A4. POST ACS,RelayState=URI&SAMLResponse=$Response(ID_{resp}, ID_{req}, DS, II_{resp}, AuthnAssert(\dots))$
S2. HTTP200 $Resource(URI)$

Fig. 2. SAML Web-browser SSO SP-Initiated

```
HTTP/1.1 302 Obj Moved\r\n
Location: IdP?SAMLRequest=AuthnReq(IS, DS, II_req, ACS, ID_req)&RelayState=URI
```

where $AuthnReq(IS, DS, II_{req}, ACS, ID_{req})$ abbreviates the XML expression:

```
<AuthnRequest ID="ID_req" Version="2.0" IssueInstant="II_req"
  Destination="DS" AssertionConsumerServiceURL="ACS"
  ProtocolBinding="HTTP-POST">
  <Issuer>IS</Issuer>
</AuthnRequest>
```

Here ID_{req} is a string uniquely identifying the request, IS is the issuer of the request, DS is the intended destination of this request, II_{req} is a timestamp, and ACS (Assertion Consumer Service) is the end-point of the SP. A common implementation choice is to use the `RelayState` field to carry the original URI that the client has requested. In step A2, C forwards the authentication request to IdP, which in turn challenges C to provide valid credentials. Note that in Figure 2 the authentication phase is abstractly represented by the dashed arrow as it is not in the scope of the SAML SSO standard. If the authentication succeeds, IdP builds the assertion $AuthnAssert(ID_{AA}, IS, II_{AA}, SJ, RC, ID_{req}, S_{ID}, NA, NB)$, where ID_{AA} is a string uniquely identifying the assertion, IS is the issuer, II_{AA} is a timestamp, SJ is the user C, RC is the intended consumer of the assertion, ID_{req} is a string uniquely identifying the request, S_{ID} is the session index, and NA and NB are `NotOnOrAfter` and `NotBefore` timestamps establishing the validity of the authentication assertion. The assertion is then included inside a SAML authentication response $Response(ID_{resp}, ID_{req}, DS, II_{resp}, AuthnAssert(\dots))$, where ID_{resp} is the ID of the response, ID_{req} the ID of the request, DS the destination, and II_{resp} is the timestamp of the operation. Then, the response is properly encoded, placed in an HTML form equipped with a self-submitting client-side script, and returned in an HTTP 200 response to the client (step A3). Finally, C transmits back the response to SP (step A4), SP checks its validity of the assertion and if these checks are successful then sends the resource to C (step S2).

3 Model Checking

We specified SAML SSO using ASLan [7], one of the specification languages developed in the context of the AVANTSSAR Project (`www.avantssar.eu`). For

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.

the sake of brevity in this paper 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)[6] and their initial state I, and ϕ is a Linear Temporal Logic (LTL) formula stating the expected security properties. (See [3] for the details). The states of M are sets of ground (i.e. variable-free) *facts*, i.e. atomic formulae of the form given in Table 1. 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. It is required that the variables occurring in R also occur in L. The rules for honest agents and the intruder are specified in Sections 3.1 and 3.2. Here and in the sequel 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`, `hReq`).

Messages are represented as follows. HTTP requests are represented by expressions $\texttt{hReq}(mtd, addr, qs, body)$, where mtd is either the constant `get` or `post`, $addr$ and qs are expressions representing the address and the query string in the URI respectively, and $body$ is the HTTP body. Similarly, HTTP responses are expressions of the form $\texttt{hRsp}(code, loc, qs, body)$, where the $code$ is either the constant `c30x` or `c200`, loc and qs are (in case of redirection) the location and the query string of the location header respectively, and $body$ is the HTTP body. In case of empty parameters, the constant `nil` is used. For instance, the message A1 in Figure 2 is `hRsp(c30x, IdP, hBind(aReq(SP, IdP, id(N)), URI), nil)` obtained by composing `hRsp`, `hBind` and `aReq`. `id(N)` is the unique ID of the request, `hBind` binds the `SAMLRequest` `aReq` and the `RelayState` `URI` to the location header. All the other HTTP fields are abstracted away because they are either not relevant for the analysis or not used by SAML SSO protocol.

[6] 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.

3.1 Specification of the rules of the honest agents

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

$$\begin{aligned}\mathtt{sent}(b_{rs}, b_i, a, m_i, c_i)\,\mathbf{.}\,\mathtt{state}_r(j, a, [e_1, \dots, e_p]) &\xrightarrow{\mathtt{send}_r^{j,k}(a,\dots)} \\ &\mathtt{sent}(a, a, b_o, m_o, c_o)\,\mathbf{.}\,\mathtt{state}_r(l, a, [e'_1, \dots, e'_q])\end{aligned} \tag{1}$$

for all honest principals a and suitable terms b_{rs}, b_i, b_o, c_i, c_o, $e_1, \dots, e_p$, $e'_1, \dots, e'_q$, m_i, m_o, and $p, q, k \in \mathbb{N}$. Rule (1) states that if principal a playing 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. Notice that, in the initial and final rules of the protocol, the fact $\mathtt{sent}(\dots)$ is omitted in the left and right hand sides of the rule (1), respectively. For instance, the reception of the message A1 in Figure 2 by the client and the sending of the message A2 are modeled by the following rewrite rule:

$$\begin{aligned}&\mathtt{sent}\,(\mathtt{SP1}, \mathtt{SP}, \mathtt{C}, \mathtt{hRsp}(\mathtt{c30x}, \mathtt{IdP}, \mathtt{AReq}, \mathtt{nil}), \mathtt{C}_{\mathtt{SP2C}})\,\mathbf{.}\\ &\quad \mathtt{state}_{\mathtt{c}}(2, \mathtt{C}, [\mathtt{SP}, \mathtt{IdP}, \mathtt{URI}, \mathtt{C}_{\mathtt{C2SP}}, \mathtt{C}_{\mathtt{SP2C}}, \mathtt{C}_{\mathtt{C2SP}_2}, \mathtt{C}_{\mathtt{SP2C}_2}, \mathtt{C}_{\mathtt{C2IdP}}, \mathtt{C}_{\mathtt{IdP2C}}])\\ &\quad \xrightarrow{\mathtt{send}_{\mathtt{c}}^{2,1}(\mathtt{C},\mathtt{IdP},\mathtt{SP},\mathtt{SP1},\mathtt{URI},\mathtt{AReq},\mathtt{C}_{\mathtt{C2SP}},\mathtt{C}_{\mathtt{SP2C}},\mathtt{C}_{\mathtt{C2SP}_2},\mathtt{C}_{\mathtt{SP2C}_2},\mathtt{C}_{\mathtt{C2IdP}},\mathtt{C}_{\mathtt{IdP2C}})}\\ &\mathtt{state}_{\mathtt{c}}(4, \mathtt{C}, [\mathtt{SP}, \mathtt{IdP}, \mathtt{URI}, \mathtt{AReq}, \mathtt{C}_{\mathtt{C2SP}}, \mathtt{C}_{\mathtt{SP2C}}, \mathtt{C}_{\mathtt{C2SP}_2}, \mathtt{C}_{\mathtt{SP2C}_2}, \mathtt{C}_{\mathtt{C2IdP}}, \mathtt{C}_{\mathtt{IdP2C}}])\,\mathbf{.}\\ &\qquad \mathtt{sent}\,(\mathtt{C}, \mathtt{C}, \mathtt{IdP}, \mathtt{hReq}(\mathtt{get}, \mathtt{IdP}, \mathtt{AReq}, \mathtt{nil}), \mathtt{C}_{\mathtt{C2IdP}})\end{aligned} \tag{2}$$

3.2 Specification of the rules of the intruder

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

$$\begin{aligned}\mathtt{sent}(\mathtt{A}, \mathtt{A}, \mathtt{B}, \mathtt{M}, \mathtt{C}) &\xrightarrow{\mathtt{intercept}(\mathtt{A},\mathtt{B},\mathtt{M},\mathtt{C})} \mathtt{ik}(\mathtt{M})\\ \mathtt{sent}(\mathtt{A}, \mathtt{A}, \mathtt{B}, \mathtt{M}, \mathtt{C}) &\xrightarrow{\mathtt{overhear}(\mathtt{A},\mathtt{B},\mathtt{M},\mathtt{C})} \mathtt{ik}(\mathtt{M})\,\mathbf{.}\,\mathit{LHS}\end{aligned} \tag{3}$$

where *LHS* is the set of facts occurring in the left hand side of the rule.

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 [21]. The decomposition capabilities of the intruder are modeled by the following rules:

$$\mathtt{ik}(\{\mathtt{M}\}_k)\,\mathbf{.}\,\mathtt{ik}(k^{-1}) \xrightarrow{\mathtt{decrypt}(\mathtt{M},\dots)} \mathtt{ik}(\mathtt{M})\,\mathbf{.}\,\mathit{LHS} \tag{4}$$

$$\mathtt{ik}(\{\mathtt{M}\}^s_{\mathtt{K}})\,\mathbf{.}\,\mathtt{ik}(\mathtt{K}) \xrightarrow{\mathtt{sdecrypt}(\mathtt{K},\mathtt{M})} \mathtt{ik}(\mathtt{M})\,\mathbf{.}\,\mathit{LHS} \tag{5}$$

$$\mathtt{ik}(f(\mathtt{M}_1, \dots, \mathtt{M}_n)) \xrightarrow{\mathtt{decompose}_f(\mathtt{M}_1,\dots,\mathtt{M}_n)} \mathtt{ik}(\mathtt{M}_1)\,\mathbf{.}\dots\mathbf{.}\,\mathtt{ik}(\mathtt{M}_n)\,\mathbf{.}\,\mathit{LHS} \tag{6}$$

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

For the composition rules we consider an optimisation [18] based on the observation that most of the messages generated by a DY intruder are rejected by the receiver as non-expected or ill-formed. Thus we restrict these rules so that the intruder sends only messages matching the patterns expected by the receiver [6]. For each protocol rule (1) in Section 3.1 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

$$\texttt{ik}(m_{1,l})\,\texttt{.}\ldots\texttt{.}\,\texttt{ik}(m_{j_l,l})\,\texttt{.}\,\texttt{state}_r(j,a,[e_1,\ldots,e_p]) \xrightarrow{\texttt{impersonate}_r^{j,k,l}(\ldots)} \texttt{sent}(\texttt{i},b_i,a,m',c_i)\,\texttt{.}\,\texttt{ik}(m')\,\texttt{.}\,\mathit{LHS} \quad (7)$$

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'.

3.3 Specifying the authentication property

The language of LTL we consider uses facts as atomic propositions, the propositional connectives (namely, $\neg$, $\vee$, $\wedge$, $\Rightarrow$), the first-order quantifiers $\forall$ and $\exists$, and the temporal operators $\mathbf{F}$ (eventually), $\mathbf{G}$ (globally), and $\mathbf{O}$ (once). Informally, given a formula ϕ, $\mathbf{F}\,\phi$ ($\mathbf{O}\,\phi$) holds if at some time in the future (past, resp.) ϕ holds. $\mathbf{G}\,\phi$ holds if ϕ always holds on the entire subsequent path. (See [3] for more details about LTL.) We use $\forall(\phi)$ and $\exists(\phi)$ as abbreviations of $\forall X_1.\ldots\forall X_n.\phi$ and $\exists X_1.\ldots\exists X_n.\phi$ respectively, where $X_1,\ldots,X_n$ are the free variables of the formula ϕ. We base our definition of authentication on Lowe's notion of *non-injective agreement* [19]. Thus, `SP` *authenticates* `C` *on* `URI` amounts to saying that whenever `SP` completes a run of the protocol apparently with `C`, then *(i)* `C` has previously been running the protocol apparently with `SP`, and *(ii)* the two agents agree on the value of `URI`. This property can be specified by the following LTL formula:

$$\mathbf{G}\,\forall(\texttt{state}_{\texttt{sp}}(7,\texttt{SP},[\texttt{C},\ldots,\texttt{URI},\ldots]) \Rightarrow \exists\,\mathbf{O}\,\texttt{state}_{\texttt{c}}(2,\texttt{C},[\texttt{SP},\ldots,\texttt{URI},\ldots])) \quad (8)$$

stating that, if `SP` reaches the last step 7 believing to talk with `C`, who requested `URI`, then sometime in the past `C` must have been in the state 2, in which he requested `URI` to `SP`.

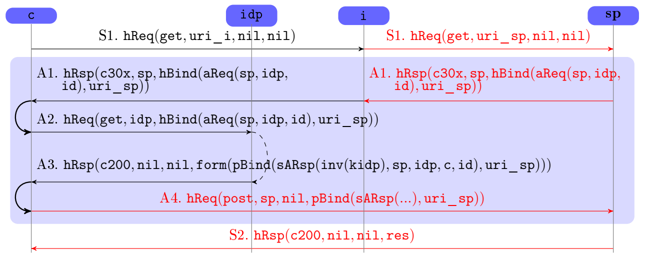
Since we aim at testing implementations using attack traces as test cases with the purpose of detecting a violation of the authentication property, we would like to be sure that at the end of the execution of the attack trace, the property has been really violated. Thus, we need to take into account the testing scenario in terms of the observability of channels and of the internal states of each principal.

This can be done by defining a set of observable facts. For instance, in case the tester can observe the messages passing through a channel `c` then, for all rs, b, a, and m, the $\mathtt{sent}(rs, b, a, m, \mathtt{c})$ facts are observable. Similarly, in case the tester can observe the internal state of an agent `a`, then for all r, j, e_1, ..., e_n the $\mathtt{state}_r(j, \mathtt{a}, [e_1, \ldots, e_n])$ facts are observable.

Once defined the set of observable facts according to the testing scenario, we rewrite the formula using them. For instance, let us suppose that the internal state of `sp` is not observable, while the channel $\mathtt{c}_{\mathtt{SP2C}}$ is observable, we rewrite the property (8) as follows:

$$\mathbf{G}\,\forall(\mathtt{sent}(\mathtt{SP}, \mathtt{SP}, \mathtt{C}, \mathtt{res}(\mathtt{URI}), \mathtt{c}_{\mathtt{SP2C}}) \Rightarrow \exists\,\mathbf{O}\,\mathtt{state}_{\mathtt{c}}(2, \mathtt{C}, [\mathtt{SP}, \ldots, \mathtt{URI}, \ldots])) \quad (9)$$

where `res(URI)` represents the resource returned by `SP` in step 7.


c
idp
i
sp
S1. hReq(get, uri_i, nil, nil)
S1. hReq(get, uri_sp, nil, nil)
A1. hRsp(c30x, sp, hBind(aReq(sp, idp, id), uri_sp))
A1. hRsp(c30x, sp, hBind(aReq(sp, idp, id), uri_sp))
A2. hReq(get, idp, hBind(aReq(sp, idp, id), uri_sp))
A3. hRsp(c200, nil, nil, form(pBind(sARsp(inv(kidp), sp, idp, c, id), uri_sp)))
A4. hReq(post, sp, nil, pBind(sARsp(...), uri_sp))
S2. hRsp(c200, nil, nil, res)
Fig. 3. Authentication Flaw of the SAML 2.0 Web Browser SSO Profile

When the model does not satisfy the expected security property, a counterexample (i.e. an *attack trace*) is generated and returned by the model checker. A violation of the authentication property (9), as discussed in [4], is witnessed by the attack depicted in Figure 3. The attack involves four principals: a client (`c`), an honest IdP (`idp`), an honest SP (`sp`), and a malicious service provider (`i`) and comprises the following steps: `c` initiates the protocol by requesting a resource `uri_i` at the SP `i`; `i`, pretending to be `c`, requests a different resource `uri_sp` at `sp` and `sp` reacts by generating an Authentication Request, which is then returned to `i`; `i` maliciously replies to `c` by sending an HTTP redirect response to `idp` containing `aReq(sp, idp, id)` and `uri_sp` (instead of `aReq(i, idp, id_i)`, and `uri_i` as the standard would mandate); the remaining steps proceed according to the standard. The attack makes `c` consume a resource from `sp`, while `c` originally asked for a resource from `i`.

4 Instrumentation

The model instrumentation is aimed at instructing 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 5) 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, messages in the formal model are specified abstractly. For instance, a SAML request $\mathit{AuthnReq}(IS, DS, II_{req}, ACS, ID_{req})$ is modeled by the expression $\texttt{aReq}(\texttt{SP}, \texttt{IdP}, \texttt{ID})$ thereby abstracting II_{req}. A further abstraction step is done by modeling two fields such as IS and ACS with only one variable $\texttt{SP}$. Let D be the set of data values the messages exchanged and their fields. For instance, if $\mathit{AuthnReq}(is, ds, ii, acs, id)$ is an element in D, then also id, ds, ii, acs, and id are in D. 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 $\alpha(\overline{f}(d_1, \dots, d_n)) = f(\alpha(d_1), \dots, \alpha(d_n))$ for all $d_1, \dots, d_n \in D$;
Selectors: $\pi^i_f : D \to D^{\perp}$ such that $\pi^i_f(d) = d_i$ if $d = \overline{f}(d_1, \dots, d_n)$ and $\pi^i_f(d) = \perp$ otherwise, for $i = 1, \dots, n$.

with the following exceptions. With $K \subseteq D$ we denote the set of cryptographic keys. If $k \in K$, then $\mathit{inv}(k)$ 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 $\mathit{decrypt}(\mathit{inv}(k), d') = d$ if $d' = \mathit{encrypt}(k, d)$ and $\mathit{decrypt}(\mathit{inv}(k), d') = \perp$ otherwise.

If $f = \texttt{senc}$, *sdecrypt* is defined similarly as above, replacing $\mathit{inv}(k)$ with k. We assume that the Adapter provides constructors and selectors as program procedures. The association between symbols and procedures are specified in the Mapping (See Figure 1).

In the specification of security protocols, the behavior of the 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. Outgoing messages are calculated without specifying which operations are performed to compute it. Therefore, in order to interact with a system under test, we need to make explicit these procedures. We write these procedures as well as the TEE in a pseudolanguage composed of statements such as *if-then-else*, *foreach*, and the like. We also assume that the pseudolanguage has a procedure $\mathrm{eval}(p)$ in order to evaluate a program fragment p. Let e be a ground expression in E. We call ℓ_e a memory location in which a data value $d \in D$ is stored such that $e = \alpha(d)$.

A data value d could be the result of the evaluation of a program fragment p, i.e. $d = \mathrm{eval}(p)$. For the sake of simplicity, in the sequel we sometimes use indifferently the data value notation and the memory location containing it. 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 the first data value available in the pipe without removing it (i.e. ℓ_c `|>` ℓ_m). The use of the latter operator will be clear to the reader when we explain the Instrumentation for the intruder's rules.

4.1 Instrumentation of the rules of the honest agents

Let us consider the following example of ASLan rule:

$$\begin{aligned}&\mathtt{sent}\,(\mathtt{A},\mathtt{A},\mathtt{B},\mathtt{f}(\{\mathtt{g}(\mathtt{A},\mathtt{B},\mathtt{m})\}^{\mathtt{s}}_{\mathtt{K}},\{\mathtt{h}(\mathtt{A},\mathtt{K})\}_{\mathtt{Kb}}),\mathtt{C_{A2B}})\,\mathtt{.}\\ &\qquad\mathtt{state_b}(1,\mathtt{B},[\mathtt{B},\mathtt{Kb},\mathtt{inv}(\mathtt{Kb}),\mathtt{m},\mathtt{C_{A2B}},\mathtt{C_{B2A}}])\xrightarrow{\mathtt{send}^{1,1}_{\mathtt{b}}(\mathtt{B},\mathtt{A},\mathtt{Kb},\mathtt{K},\mathtt{C_{A2B}},\mathtt{C_{B2A}})}\\ &\qquad\qquad\mathtt{state_b}(2,\mathtt{B},[\ldots,\mathtt{A},\mathtt{K}])\,\mathtt{.}\,\mathtt{sent}\,(\mathtt{B},\mathtt{B},\mathtt{A},\mathtt{f}(\mathtt{B},\mathtt{m}),\mathtt{C_{B2A}})\end{aligned}\quad(10)$$

This rule can be executed only if the message received on the channel $\ell_{C_{A2B}}$ is $\overline{\mathtt{f}}(d_1, d_2))$, where d_1 can be decrypted only after having decrypted d_2, containing the data value of the decryption key K. Moreover d_1 must be $\overline{\mathtt{g}}(d_3, d_4, d_5))$, where d_3 is simply stored in ℓ_A, while d_5 must be equal to $\ell_{\mathtt{m}}$, and d_4 must be equal to ℓ_B, given that the variables B belongs to the internal state of the agent. As said, these checks are implicit in the ASLan semantics (pattern matching), as well as the procedure necessary to construct the message $\ell_{\mathtt{f}(B,\mathtt{m})}$, which is sent on the channel $\ell_{C_{B2A}}$. Nevertheless, for the testing purpose, we need to explicit these procedures. They only depend on the structure of the rule and thus can be precomputed. A program fragment $p_{\mathtt{send}^{j,k}_r(a,\ldots,c_i,c_o)}$ encoding a rule (1) is as follows:

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

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 $m_o = \alpha(\ell_{m_o})$. In the sequel, we describe how to generate automatically p_{m_i} and p_{m_o} for a generic ASLan rule (1).

We define an association between an ASLan expression e and the fragment p used to retrieve –accessing directly to memory locations or using selectors operating on them– the corresponding data value denoted by e. 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 $e = \alpha(\text{eval}(p))$.

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.

Definition 1 (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* $(decrypt(p_2,p_1) : e) \in \downarrow Ms$,
3. *if* $p_1 : \mathtt{senc}(k,e) \in \downarrow Ms$ *and* $p_2 : k \in \downarrow Ms$, *then* $(sdecrypt(p_2,p_1) : e) \in \downarrow Ms$,
4. *if* $p : f(e_1,\ldots,e_n) \in \downarrow Ms$, *then* $(\pi^j_f(p) : e_j) \in \downarrow Ms$ *for* $j = 1,\ldots,n$.

Let us provide an example of closure. With reference to the rule (10), the set Ms contains the associated expression for the incoming message $\ell_{\mathtt{f}(\mathtt{senc}(\ldots),\mathtt{enc}(\ldots))} : \mathtt{f}(\mathtt{senc}(K,\mathtt{g}(A,B,\mathtt{m})),\mathtt{enc}(Kb,\mathtt{h}(A,K)))$ and other expressions known by the agent $\ell_B : B$, $\ell_{Kb} : Kb$, $\ell_{\mathtt{inv}(Kb)} : \mathtt{inv}(Kb)$, $\ell_{\mathtt{m}} : \mathtt{m}$, $\ell_{C_{A2B}} : C_{A2B}$, and $\ell_{C_{B2A}} : C_{B2A}$. By definition $\downarrow Ms$ contains Ms and other associated expressions. For example, we have $\ell_{\mathtt{f}(\mathtt{senc}(\ldots),\mathtt{enc}(\ldots))} : \mathtt{f}(\mathtt{senc}(\ldots),\mathtt{enc}(Kb,\mathtt{h}(A,K))) \in Ms \subseteq \downarrow Ms$ then $\pi^1_{\mathtt{f}}(\ell_{\mathtt{f}(\mathtt{senc}(\ldots),\mathtt{enc}(Kb,\mathtt{h}(A,K)))}) : \mathtt{senc}(\ldots)$ and $\pi^2_{\mathtt{f}}(\ell_{\mathtt{f}(\mathtt{senc}(\ldots),\mathtt{enc}(Kb,\mathtt{h}(A,K)))}) : \mathtt{enc}(Kb,\mathtt{h}(A,K))$ are in $\downarrow Ms$ (case 4 of the definition). Given that $\ell_{Kb} : Kb$ is in $\downarrow Ms$, the case 2 is applicable, thus $decrypt(\ell_{\mathtt{inv}(Kb)},\pi^2_{\mathtt{f}}(\ldots)) : \mathtt{h}(A,K) \in \downarrow Ms$ as well. The example can be easily extended to the other sub-terms of the message. However, it already clarifies why we need the closure of the knowledge. Indeed, the first part of the message $\mathtt{f}(\ldots)$ is encrypted with K and it can be decrypted only after having decrypted the second part, containing the key K. Notice that, for the sake of simplicity, in this paper 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 $Ms' = \downarrow Ms - \{\ell_{e_1} : e_1,\ldots,\ell_{e_n} : e_n\}$.

Definition 2 (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;`

For instance, let us consider the rule (10), the following checks are in $P_{\mathtt{f}(\ldots)}$:

1. `if` eval$(\pi^3_{\mathtt{g}}(sdecrypt(\pi^2_{\mathtt{h}}(\ldots),\pi^1_{\mathtt{f}}(\ldots))))$ `!=` $\ell_{\mathtt{m}}$ `then: return false;`
   `if` eval$(\pi^2_{\mathtt{g}}(sdecrypt(\pi^2_{\mathtt{h}}(\ldots),\pi^1_{\mathtt{f}}(\ldots))))$ `!=` ℓ_B `then: return false;`
2. ℓ_A `:=` eval$(\pi^1_{\mathtt{h}}(decrypt(\ell_{\mathtt{inv}(Kb)},\pi^2_{\mathtt{f}}(\ldots))))$;
   `if` (ℓ_A`!=`eval$(\pi^1_{\mathtt{g}}(sdecrypt(\pi^2_{\mathtt{h}}(\ldots),\pi^1_{\mathtt{f}}(\ldots))))$ `) then: return false;` ...

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

Definition 3 (Message generation function). *We call* message generation function *over a set of expressions kn a function* MsgGen *defined as follows:*

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

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

4.2 Instrumentation of the rules of the intruder

Intercept and overhear rules Let us consider the intercept rule (4) in Section 3. Let M be the message. The fragment $p_{\texttt{intercept}(A,B,M,C)}$ of pseudocode encoding the rule is as follows:

```
$\ell'_M$ := $\ell_M$;
$\ell_c$ >> $\ell_M$;
if $\ell'_M$ is not empty and $\ell_M$ != $\ell'_M$ 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 (4) in Section 3 is the same as the one defined above, except from the operator |> in place of >>.

Decomposition rules Let us consider the rules modeling the ability of decomposing messages (i.e. decrypt, sdecrypt, and decompose).

The fragment of pseudocode $p_{\texttt{decrypt}(M,\ldots)}$ encoding the rule (4) is as follows:

```
$\ell_M$ := eval($decrypt(\ell_{\texttt{inv}(K)}, \ell_{\{M\}_K})$);
```

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 (5).

The fragment $p_{\texttt{decompose}_f(M_1,\ldots,M_n)}$ encoding the rule (6) is as follows:

```
$\ell_{M_1}$ := eval($\pi^1_f(\ell_{f(M_1,\ldots,M_n)})$);
$\vdots$
$\ell_{M_n}$ := eval($\pi^n_f(\ell_{f(M_1,\ldots,M_n)})$);
```

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

Composition rules Let us consider the impersonate rule (7) in Section 3. The fragment of pseudocode $p_{\texttt{impersonate}^{j,k,l}_r(\ldots)}$ encoding this rule is computed by $\mathrm{MsgGen}(m')$ over the knowledge $kn = \{m_{1,l}, \ldots, m_{j_l,l}\}$.

5 Test Case Execution

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;[$step_1, \ldots, step_n$]:Attack Trace)
2  for $i$:=1 to $n$ do:
3   if not($step_i$ == $\mathsf{send}_r^{j,k}(a, \ldots)$ and $a \in SUT$) then:
4     if not $\mathrm{eval}(p_{step_i})$ 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's 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)) notifies that an error occurred.

6 Experimental Results

In order to assess the effectiveness of the proposed approach, we have developed a prototype of the architecture depicted in Figure 1.

We have implemented the Instrumentation, the TEE and the Adapter components in Java. The Model Checking module is the SATMC model checker tool [2] taken off-the-shelf from the AVANTSSAR Platform. The Instrumentation component takes an ASLan model and the Mapping as input. It produces program fragments in a Java class. The TEE instantiates the class and executes the attack trace as described in Section 5. The Adapter implements the constructor and selector functions defined in Section 4. For example, constructors and selectors for the HTTP protocol are available in a Java class called `adapter.Http` that is built upon the Apache HttpComponents (`http://hc.apache.org/`). Those for the SAML SSO protocol in a class called `adapter.Saml` that is based on OpenSAML (`https://wiki.shibboleth.net/confluence/display/OpenSAML/Home`). These functions are used by program fragments as described in Section 4.

We extended the formal model of the SAML SSO we developed in previous work [4] by modeling messages using ASLan expressions as seen in Section 3. We provided the formal model to the model checker together with the authentication property (9). The model checker found the attack trace depicted in Figure 3.

We have tested two Web-based SSO solutions freely available on-line, the SAML-based SSO for Google Apps (`http://code.google.com/googleapps/domain/sso/saml_reference_implementation.html`) and the SimpleSAMLphp SSO service offered by Foodle, a surveys and polls on-line service (`https://foodl.org`). We have specified two mappings, one for each solution. For example, the mapping for testing SAML-based SSO for Google Apps contains associations as $\overline{\texttt{uri}_{\texttt{sp}}}$ = "http://mail.google.com/a/ai-lab.it/h" and $\overline{\texttt{hReq}}$ = adapter.Http where $\overline{\texttt{uri}_{\texttt{sp}}}$, $\overline{\texttt{hReq}}$ are constructor functions.

We have run the prototype against the SAML-based SSO for Google Apps by using the set $\{\texttt{idp}, \texttt{sp}\}$ as SUT Configuration. The SP is the Google GMail service while the IdP is a local identity provider service at the AI-Lab. The TEE automatically executed the attack traces till the message S2 of Figure 3 and, as expected, the message S2 contains the mailbox of the user. Therefore, the prototype was able to automatically detect the authentication flaw.

We have used the same SUT Configuration in the experiment with SimpleSAMLphp. In this case we used Foodle as SP and Feide OpenIdP identity provider (`https://openidp.feide.no`) as IdP. The execution of the attack failed when message S2 was received. The analysis of exchanged messages has revealed that SimpleSAMLphp returns an error message instead of the message S2. We identified the cause in additional checks that reinforce the binding between authentication requests and responses. These checks are based on cookies and, since the authentication request is never routed through `c`, no cookies are installed in `c`. Therefore, when `c` presents an authentication response at `sp`, it fails in restoring the local user session for `c`.

7 Related Work

Automated analysis of security protocols has been studied and several analysis tools have been developed (see e.g., [1, 10, 24]). Also, there have been applications of model checking to the security analysis of Web Services (e.g., [8, 17, 27]). These approaches mostly focus on design time verification, and fall short in validating whether the real systems satisfy the desired properties in later life stages. Model-based testing has been applied to security-relevant systems in the recent past, e.g., [25, 26]. These approaches do not propose a coherent generic methodology for security testing. Also, mappings between the abstract and concrete levels are currently managed in an ad-hoc manner only [30].

Model-checkers have been already proposed for testing by interpreting counterexamples as test cases. (See [16] for a survey). However there is no systematic approach for execution and interpretation of counterexamples.

Security-specific mutation operators have been considered in order to introduce implementation-level vulnerabilities into models [12, 14]. These approaches focus on detecting implementation-level vulnerabilities. They extend and complete the one we presented. Indeed, when a model is secure with respect to a security property, it is mutated by using a security-specific mutation operator. Moreover, it does not only consider logical flaw but also vulnerabilities at the implementation level.

TorX is an automated model-based testing tool that aim at improving the quality of the software in an on-the-fly manner [29]. Its architecture has a module providing a connection with the SUT in order to send input and receiving output. However, more generic approaches for implementing adapters are needed.

An approach for model-checking driven security testing is proposed in [5]. Although the approach is protocol independent, it is strictly focused on the concretization of abstract messages in order to derive concrete test cases.

The automated tool Tookan [11] is based on an approach similar to the one we described. It reverse-engineers a real PKCS#11 token to deduce its functionality, constructs a model of its API for the SATMC model checker, and then executes any attack trace found by the model checker directly on the token. Nevertheless, this approach is specific for the PKCS#11 security tokens.

8 Conclusions

In this paper we proposed an approach that supports the binding of specifications of security protocols to actual implementations through model instrumentation, and the automatic testing of real implementations against putative attacks found by a model checker. The approach consists in model checking a formal model looking for a counterexample (i.e. attack trace) violating a security property. In case an attack is returned, it calculates automatically program fragments encoding how to verify and generate protocol messages. The attack trace is interpreted and the program fragments are executed accordingly.

In order to assess the effectiveness of the proposed approach we developed a prototype and used it to test two Web-based Single Sign-On (SSO) solutions that are available on-line, namely the SAML-based SSO for Google Apps and the SimpleSAMLphp SSO service offered by Foodle. The prototype is able to successfully execute an attack on the Google service. The prototype also shows that the same attack does not succeed against the SSO service of Foodle, due to specific implementation mechanisms used by SimpleSAMLphp.

Application of our techniques on other protocols (e.g. OpenID, OAuth) is under way and confirms the viability of the approach.

References

1. A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Heám, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In *Proc. of CAV05.*
2. A. Armando, R. Carbone, and L. Compagna. LTL Model Checking for Security Protocols. In *Journal of Applied Non-Classical Logics*, 2009.
3. A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. T. Abad. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In *Proc. of ACM FMSE08.*
4. A. Armando, R. Carbone, L. Compagna, J. Cuellar, G. Pellegrino, and A. Sorniotti. From multiple credentials to browser-based single sign-on: Are we more secure? In *Proc. IFIP TC SEC2011.*
5. A. Armando, R. Carbone, L. Compagna, K. Li, and G. Pellegrino. Model-checking driven security testing of web-based applications. In *Proc. of ICSTW2010.*

6. A. Armando and L. Compagna. Automatic SAT-Compilation of Protocol Insecurity Problems via Reduction to Planning. In *Proc. of FORTE 2002*.
7. AVANTSSAR. Deliverable 2.1: Requirements for modelling and ASLan v.1. Available at `http://www.avantssar.eu`, 2008.
8. M. Backes, S. Mödersheim, B. Pfitzmann, and L. Viganò. Symbolic and Cryptographic Analysis of the Secure WS-ReliableMessaging Scenario. In *Proc. of FOSSACS'06*.
9. J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010.
10. B. Blanchet. Automatic verification of cryptographic protocols: A logic programming approach (invited talk). In *Proc. of PPDP'03*.
11. M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *ACM Conf. on CSS*.
12. M. Büchler, J. Oudinet, and A. Pretschner. Security mutants for property-based testing. In *TAP 2011*.
13. J. Clark and J. Jacob. A Survey of Authentication Protocol Literature: Version 1.0, 17. Nov. 1997. URL: `www.cs.york.ac.uk/~jac/papers/drareview.ps.gz`.
14. F. Dadeau, P.-C. Héandam, and R. Kheddam. Mutation-based test generation from security protocols in HLPSL. In *ICST 2011*.
15. A. Doupé, M. Cova, and G. Vigna. Why johnny can't pentest: an analysis of black-box web vulnerability scanners. In *Proc. DIMVA 2010*.
16. G. Fraser, F. Wotawa, and P. Ammann. Testing with model checkers: a survey. *Softw. Test., Verif. Reliab. 2009*, 19.
17. M. Hondo, N. Nagaratnam, and A. Nadalin. Securing web services. *IBM Systems Journal*, 41(2):228–241, 2002.
18. F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and Verifying Security Protocols. In *Proc. of LPAR 2000*.
19. G. Lowe. A hierarchy of authentication specifications. In *Proc. of the 10th IEEE CSFW '97*.
20. G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Proc. of TACAS'96*, 1996.
21. W. Marrero, E. M. Clarke, and S. Jha. Model checking for security protocols. tech. report cmu-scs-97-139. Technical report, CMU, May 1997.
22. J. K. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proc. of ACM CCS'01*.
23. OASIS. SAML V2.0. `http://docs.oasis-open.org/security/saml/v2.0/`, 2005.
24. P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *Modelling and Analysis of Security Protocols.* Addison Wesley, 2000.
25. P. A. P. Salas, P. Krishnan, and K. J. Ross. Model-based security vulnerability testing. *Australian Software Engineering Conf.*, 0:284–296, 2007.
26. P. P. Salas and P. Krishnan. Testing privacy policies using models. In *Proc. of SEFM '08*.
27. G. Salaün, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. In *Proc. of ICWS'04*.
28. V. Shmatikov and J. C. Mitchell. Finite-state analysis of two contract signing protocols. *Theoretical Computer Science*, 283(2):419–450, June 2002.
29. G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In *First European Conf. on Model-Driven Software Engineering*.
30. M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Technical report, University of Waikato, New Zealand.