Detection of logic flaws in multi-party business applications via security testing

Giancarlo Pellegrino

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2013-ENST-0064

EDITE - ED 130

Doctorate ParisTech

T H E S I S

TELECOM ParisTech

in « Computer Science and Networks »

Giancarlo PELLEGRINO

08/11/2013

Detection of Logic Flaws in Multi-party Business Applications via Security Testing

Advisor : **Davide BALZAROTTI**

THÈSE

Jury members :

M. Engin KIRDA, Professor, Northeastern University, US	Reporter
M. Frédéric CUPPENS, Professor, Télécom Bretagne, FR	Reporter
M. Refik MOLVA, Professor, EURECOM, FR	Examiner
M. Alessandro ARMANDO, Professor, Università degli Studi di Genova, IT	Examiner
M. Luca COMPAGNA, Doctor, SAP Product Security Research, FR	Examiner

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2013-ENST-0064

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « Informatique et Réseaux »

présentée et soutenue publiquement par

Giancarlo PELLEGRINO

08/11/2013

Détection d'Anomalies Logiques dans les Logiciels d'Entreprise Multi-partis à travers des Tests de Sécurité

Directeur de thèse : **Davide BALZAROTTI**

Jury

M. Engin KIRDA, Professeur, Northeastern University, US	Rapporteur
M. Frédéric CUPPENS, Professeur, Télécom Bretagne, FR	Rapporteur
M. Refik MOLVA, Professeur, EURECOM, FR	Examinateur
M. Alessandro ARMANDO, Professeur, Università degli Studi di Genova, IT	Examinateur
M. Luca COMPAGNA, Docteur, SAP Product Security Research, FR	Examinateur

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

T H È S E

Acknowledgements

I would like first to express my gratitude to Davide Balzarotti. I have learned so much from him and this thesis would not have been possible without his supervision.

I would like to thank the following people for their inspiration, support, help, company, and encouragement: Luca C., Giampaolo, Peter, Alessandro A., Roberto Carbone, Alessandro S., Volkmar, Jean-Christophe, Sylvine, Luca Z., Andreas, Roberto Catanuto, Michele, Thomas, Stefan, Florian K., Martin, Xavier, Achim, Francesco, Antonino, Theodoor, Kuba, Slim, Antonella, Gerald, Julia, Akram, Johann, Jörn, Florian S., Stephane, Gabi, Matteo, Serena, Corentin, Samuel, the-noisy-printer, Sebastian, Cédric, Rosario, Luca M., the S3 group, the SPaCIoS group, and the AVANTSSAR group.

I would like also to give a special thank to Christina, a famiglia, Mercecindo, Saro, Tama, and Kata, to have supported me with the right combination of care and non-sense.

Finally, thanks to prof. Engin Kirda, prof. Frédéric Cuppens, prof. Refik Molva, prof. Alessandro Armando, and dr. Luca Compagna for agreeing to be reporters and examiners.

This thesis has been funded by SAP AG, by the EU funded project FP7-ICT-2009-5 no. 257876, "SPaCIoS: Secure Provision and Consumption in the Internet of Services", and the EU funded project FP7-ICT-2007-1 no. 216471, "AVANTSSAR: Automated Validation of Trust and Security of Service-oriented Architectures".

Abstract

Multi-party business applications are computer programs distributed over the Web implementing collaborative business functions. These applications are one of the main target of attackers who exploit vulnerabilities in order to perform malicious activities. The most prevalent classes of vulnerabilities are the consequence of insufficient validation of the user-provided input. However, the less-known class of *logic vulnerabilities* recently attracted the attention of researcher. Unfortunately, the unavailability of the source code in these kind of applications makes it hard to discover this type of vulnerabilities. According to the availability of software documentation, two further techniques can be used: design verification via model checking, and black-box security testing. However, the former offers no support to test real implementations and the latter lacks the sophistication to detect logic flaws.

In this thesis, we present two novel security testing techniques to detect logic flaws in multi-party business applications that tackle the shortcomings of the existing techniques. First, we present the design verification via model checking of two security protcools. We then address the challenge of extending the results of the model checker to automatically test real protocol implementations. Second, when explicit documentation is not available, we present a novel black-box security testing technique that combines model inference, extraction of workflow and data flow patterns, and an attack pattern-based test case generation algorithm. Finally, we discuss the application of the technique developed in this thesis in an industrial setting.

We used the techniques presented in this thesis to discover previously-unknown design errors in the SAML SSO and OpenID security protocols and their implementations, and ten severe logic vulnerabilities in eCommerce business applications allowing an attacker to pay less or even shop for free.

Table of Contents

List of Figures

List of Tables

List of Publications

The results of thesis have been published in peer-reviewed journals, conferences, and workshops. The list of contribution is the following:

[PB14] G. Pellegrino, D. Balzarotti, *Toward Black-box Detection of Logic Flaws in Web Applications*, Proceedings of Network and Distributed System Security Symposium 2014 (NDSS'14), San Diego, CA, 2014

[PCM13] G. Pellegrino, L. Compagna, T. Morreggia *A Tool for Supporting Developers in Analyzing the Security of Web-based Security Protocols*, 25th IFIP International Conference on Testing Software and Systems (ICTSS'13), Istanbul, Turkey, November 13-15, 2013

[ACC13] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, G. Pellegrino, A. Sorniotti, *An Authentication Flaw in Browser-based Single Sign-On Protocols: Impact and Remediations*, Computers & Security, 2013

[APC12] A. Armando, G. Pellegrino, R. Carbone, A. Merlo, D. Balzarotti, *From Model-checking to Automated Testing of Security Protocols: Bridging the Gap*, 6th International Conference on Tests & Proofs (TAP 2012), Prague (Czech Republic), May 31 - June 1, 2012

[AAA12] A. Armando, W. Arsac, T. Avanesov, M. Barletta, A. Calvi, A. Cappai, R. Carbone, Y. Chevalier, L. Compagna, J. Cuéllar, G. Erzse, S. Frau, M. Minea, S. Mödersheim, D. von Oheimb, G. Pellegrino, S. E. Ponta, M. Rocchetto, M. Rusinowitch, M. Torabi Dashti, M. Turuani, and L. Viganó. *The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures*, 18th International Conference

on Tools and Algorihms for the Construction and Analysis of Systems (TACAS 2012), Talling, Estonia, March 24 - April 1, 2012

[ACC11] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, G. Pellegrino, A. Sorniotti, *From Multiple Credentials to Browser-based Single Sign-On: Are We More Secure?*, 26th IFIP TC-11 International Information Security Conference (SEC 2011), Luzern (Switzerland), June 7-9, 2011

[ACCP11] A. Armando, R. Carbone, L. Compagna, G. Pellegrino, *Automatic security analysis of SAML-based single sign-on protocols*, Chapter 10 in "Digital Identity and Access Management: Technologies and Framework", Business Science. Editors: Raj Sharman R., Das Smith S., Gupta M., December 2011

[ACC10] A. Armando, R. Carbone, L. Compagna, K. Li, G. Pellegrino, *Model-checking Driven Security Testing of Web-Based Applications*, International Workshop on Modeling and Detection of Vulnerabilities (MDV 2010), Paris (France), April 10, 2010

Chapter 1
Introduction

1.1 Multi-party Business Applications

1.1.1 Historical Outline

Business applications are computer programs that are used to perform business functions. The way business applications are developed, deployed and consumed faced fundamental changes over the last decades.

Business applications appeared for the first time in the 1960s. Their architecture was monolithic, in which the data management, the application logic, and the presentation were implemented in the same code. Business applications were deployed in room-size mainframes and accessed via terminals. In the 1970s, data management was isolated into an independent component originating two-tier software architectures. Both tiers were still deployed in mainframes. In the 1980s, three-tier architectures replaced two-tiers, in which the presentation was separated from the application logic. The new presentation layer was implemented in independent applications that were deployed on workstations and personal computers, while the data and application logic functions remained on the mainframe. Clients and servers were within the premises of the organization and connected to each other through local networks. In the 1990s, the architecture of business applications evolved into a multi-tier architecture in which data management and application logic were distributed over several servers. Applications were

still accessed using the client-server paradigm, however servers were located in different sites of the same organization.

Nowadays, business applications are developed as a composition of services. Each service implements a basic business function, and can be deployed over the network. Business applications are accessed via web browsers or web-enabled client applications running on personal computers, or mobile devices. While in the past business applications were available within a private network, nowadays they can be accessed from public networks such as the Internet.

1.1.2 Collaboration in Modern Business

Collaboration between organizations is fundamental for modern businesses to remain competitive [Xu07, KMR05, DHL01]. It can be implemented by meshing the business processes of an organization with the business processes of its customers, suppliers, and other business partners [XB05].

Figure 1.1 shows an example of the *procurement process*. This process is used throughout this manuscript as a running example. In general, the term procurement refers to the purchase of goods and services from external entities for satisfying a need of the buyer. The process involves several steps, e.g., the identification of the need of the buyer, the identification of the supplier, ordering, payment, and billing. In our example we consider a simplified version with three steps: ordering goods, payment, and billing.

The procurement process of Figure 1.1 involves three business partners. They are Buyer Inc., Seller Inc., and Bank Inc.. The scenario originates from the need of Buyer to purchase goods and services. Seller is a supplier specialized in business-to-business provisioning, and Bank Inc. manages the monetary transaction between Buyer and Seller. The process is described from the point of view of Buyer Inc. and proceeds as follows. First, the employee U of Buyer Inc. accesses the store of Seller Inc. and chooses the good to purchase in the catalogue of Seller Inc.. Then, U authorizes the bank to transfer the amount of money of the goods to the account of the Seller Inc. Finally, U receives a notification from Seller confirming the purchase

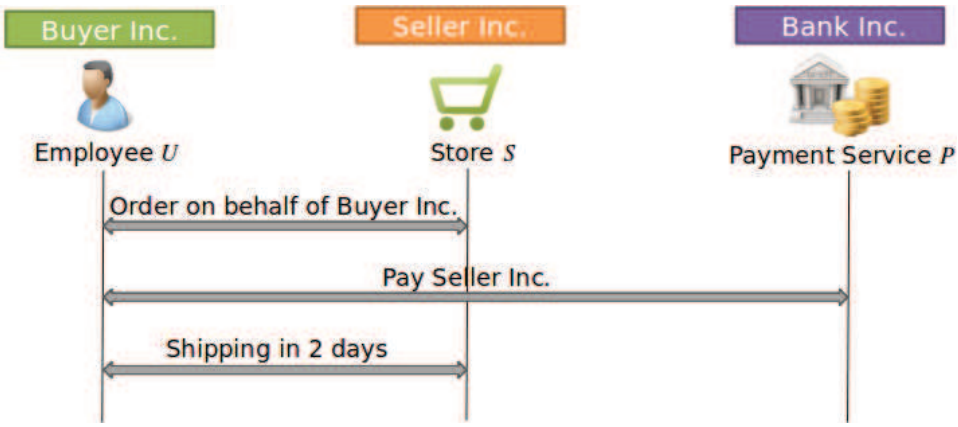


Figure 1.1: Reference business scenario: the procurement process

with the details of the delivery.

As shown in Figure 1.1, the procurement process can be implemented as a collection of services. Seller owns a platform implementing an online catalog, a virtual shopping cart, and a customer care service. The platform is available on the Internet as a service, say S. The Bank Inc. offers an online payment service P for performing monetary transactions and payment via credit cards. Finally, Buyer uses a web application that composes these services together implementing the process in Figure 1.1.

The logic of the application in Figure 1.1 can be defined in terms of expectations of the business partners. For example, at the end of the execution, the parties involved have the following expectations:

1. Seller Inc. and Buyer Inc. agreed on the goods of the purchase and on their price;

2. The bank transferred the amount of money from an account of Buyer Inc. to the account of Seller Inc.;

3. Seller Inc. delivers the goods to Buyer Inc.

Moreover, Buyer Inc. has additional security requirements, i.e., only authorized employees working for the procurement department can create

orders at S and authorize payments at P. These type of requirements can be satisfied by using security protocols.

1.1.3 The Role of Security Protocols

Prior to computer networks, security of business applications was achieved by restricting physical access to the mainframes [MCJ97]. Physical restriction was then replaced by authentication schemes in which a user provides username and password over a terminal [MCJ97]. This mechanism was sufficient to access the computer via secure communication channels [MCJ97]. In 1980s, the need of sharing data and computational resources led to the creation of computer networks in which organizations connected computers to each other over insecure communication channels. As a result, attackers could intercept and reuse user credentials to gain unauthorized access. Exchanging credentials in clear-text over these links was no longer a secure practice [MCJ97]. This led to the development of user authentication protocols and other security protocols based on cryptographic primitives (See the Security Protocols Open Repository [spo02]).

Security protocols play two roles in modern business applications. First, they provide the security guarantees that business applications need in order to carry out the business functions, e.g., user authentication and confidential message exchange. Second, they are enabling technology for business collaborations. For example, security protocols allow business partners to set up federated identity management, or enable an organization to share resources with partners keeping the ownership and the access control.

With reference to Figure 1.1, let us consider the following two security-relevant requirements. First, Buyer Inc. would like that only authorized employees of the procurement department can create orders at S, and authorize payments at P. Second, Buyer Inc. would like that its employees are authenticated only once to access the services S and P.

These two requirements are satisfied by security protocols. For example, Figure 1.2 shows a detailed description of the procurement process that extends Figure 1.1 by adding two new services IdP and AS. IdP is the identity

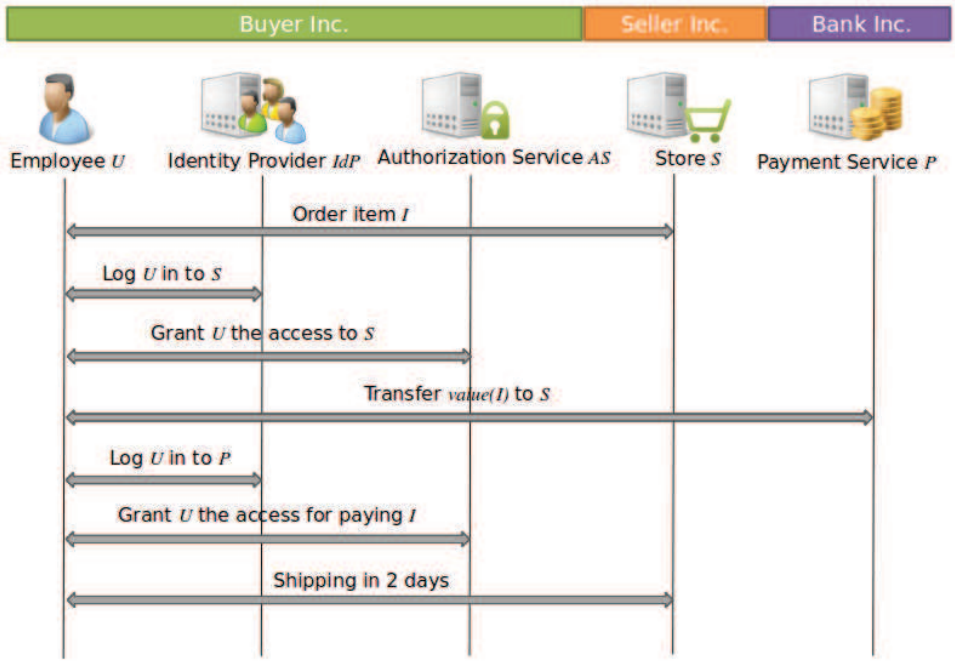


Figure 1.2: A detailed view of the procurement process

management provider in charge of authenticating users of the organization of Buyer Inc.. AS is the authorization service that grants or denies the access to resources to the employees of Buyer Inc.. The user authentication and authorization are performed upon a request issued by a service called initiator. Then, IdP, or AS, issues a signed token for the initiator that proves that the user is authenticated, or authorized, respectively. For the sake of simplicity, in Figure 1.2 we omit the token generation and exchange.

The process is the following. First, U accesses S for making an order for the item I. S does not known the identity of U nor whether U is authorized to make orders. Thus, S asks U to be authenticated by IdP, the identity provider of his own company, and to be authorized by AS to make an order at S. U shows to S two messages signed by IdP and AS to prove that she is authenticated and authorized. Afterwards, U confirms the order and visits P for transferring money to the account of the store. Similarly as seen before, P does not know the identity of U nor whether U is authorized to access P.

So, P asks U to be authenticated at IdP and to get the authorization for paying S. Afterwards, P pays Seller Inc. the value of I as asked by U. Finally, U receives a notification from S confirming the purchase with the details of the delivery.

1.2 Security Risks of Multi-party Business Applications

Multi-party business applications play a very important role in many areas, and are currently trusted by billions of users and companies to purchase goods and services, perform financial transactions, and store confidential data. Unfortunately, this makes these applications one of the primary targets for attackers interested in a wide range of malicious activities.

As seen in the previous sections, the surface of business applications accessible to external entities increased over the years. On the one hand, the software architecture shifted from monolithic and centralized to multi-tier and distributed. On the other hand, the IT infrastructure changed from a mainframe-terminals to a client-server architecture. As a result, software vulnerabilities are no longer visible only to the member of an organization, but also to external actors. Therefore, the risk of being attacked by external actors dramatically increased.

1.2.1 Threats to Multi-party Business Applications and Economical Impact

According to the Verizon Data Breach Investigations Report 2013, there are three types of cyber threats to an organization: internal actors, partners, and external actors [Ver13]. An *internal actor* is a person that works for the organization. For example, the employee U is an internal actor of Buyer Inc. A *partner* is a business partner. For example, employees of Buyer Inc. and employees of Seller Inc. are partners of each other. An *external actor* is a person outside of both the organization and partners.

In the last five years the number and frequency of attacks from external

actors has increased dramatically. According to the Verizon Data Breach Investigations Report 2013, attacks from external actors increased by 17.8% over the last 5 years (from 78% in 2008 up to 92% in 2012) [Ver13]. On the contrary, the attacks from internal actors decreased by 64.1% (from 39% in 2008 to 4% in 2012) [Ver13] while attacks from partners have been always relatively low (5% in 2008 and 1% in 2012 [Ver13]). According to the Cost of Cyber Crime Study by the Ponemon Institute, the average number of successful attacks in 2012 is 1.8 per week with an average growth by about 40% each year [Pon12]. Moreover, the Ponemon Institute reported that in the same year, 64% of the companies experienced at least one attack coming from the Web [Pon12].

According to the Verizon Data Breach Investigations Report 2013, most of the external attacks are coming from organized criminal groups, State-affiliated groups, independent groups, activists, and former employees [Ver13]. About 70% of attacks are performed by criminal and State-affiliated groups (50% and 20% respectively) [Ver13]. The majority of the attacks have financial motivations such as payment fraud, and identity theft [Ver13].

The economical impact of web-based attacks is still very high [Pon12]. In 2012, each company had an average cost of $8.9 million due to the cyber attacks with an average increment of about +$0.5 million from 2011 and about +$2,45 million from 2010 [Pon12]. The total cost is calculated considering indirect costs such as the costs for the detection, investigation, containment, and recovery, and direct costs due to the information loss, costs for business disruption, equipment damage, and revenue loss [Pon12]. The *average cost of web-based attacks is about $1 million per organization in 2012* [Pon12].

1.2.2 The Rise of Logic Flaws

The most prevalent class of web applications vulnerabilities is due to insufficient validation of user input, e.g., SQL injection (SQLi) and Cross-Site Scripting (XSS) [MIT]. This type of vulnerabilities has been largely studied by the scientific community [HVO06, SBK12]. Another, less known, class of web vulnerabilities that only recently has attracted the attention of re-

searchers is the one related to logic errors.

Logic vulnerabilities still lack a formal definition but, in general, they are often the consequence of an improper validation of the business process of an application. The resulting violations may involve both the control plane (i.e., the navigation between different pages) and the data plane (i.e., the data flow that links together parameters of different pages). In the first case, the root cause is the fact that the application fails to properly enforce the sequence of actions performed by the user. For example, it may not require a user to log in as administrator before changing the database settings (authentication bypass), or it may not check that all the steps in the checkout process of a shopping cart are executed in the right order. Logic errors involving the data flow of the application are instead caused by failing to enforce that certain values, which are propagated between different HTTP requests, are not modified by the user. For example, an attacker can try to replay expired authentication tokens, or mix together the values obtained by running several parallel sessions of the same web application.

Logic vulnerabilities can be seen also from the perspective of testing. The goal of security testing is to find an execution of the software under test that proves the existence of a vulnerability. The ability to find this particular type of execution depends from two factors. First, the tester must be able to generate the proper executions and second, the tester must be able to decide whether an execution is proving the presence of a vulnerability. Both abilities may or may not need models of the application. For example, to detect stored XSS, a tester needs a behavioral model for the test case generation whereas she does not need a model to decide if an execution detected a flaw. Conversely, to detect logic flaws, the tester may not need a model for test generation however she needs a model of the application logic to decide whether the execution proves the existence of a vulnerability. From this perspective, the importance of models in order to discover logic vulnerabilities emerges.

In the last years, we have observed an increasing number of security incidents due to logic flaws. Figure 1.3 shows the number of incidents that are reported in the Common Vulnerability Enumeration database (CVE-

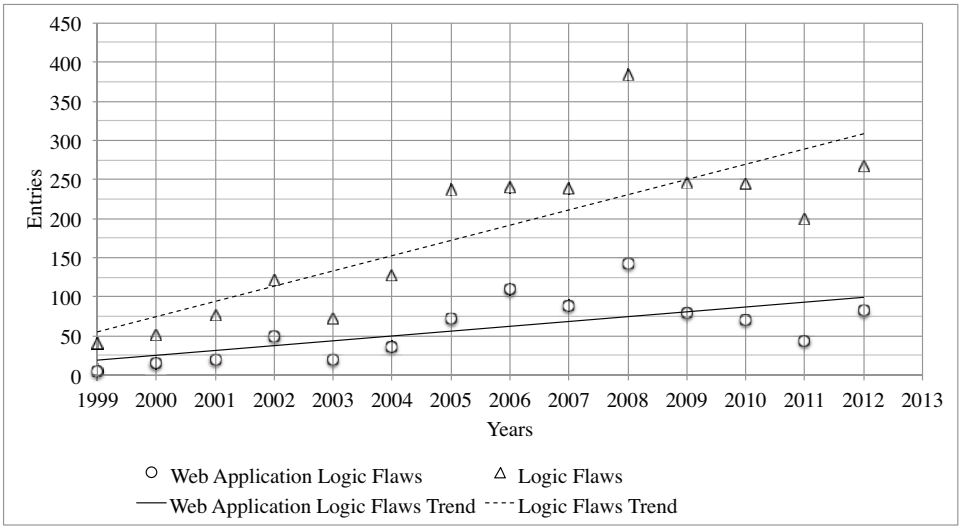


Figure 1.3: Security Incidents due to Logic Flaws from 1999 to 2012

DB), and that are caused by logic flaws from the year 1999 till the year 2012[1]. As can be seen in Figure 1.3, the general trend of incidents due to logic flaws is increasing over the years. The number of these incidents in 2012 is 267, 82 of which in web applications. The highest peak is in 2008 with 384 incidents, 143 of which in web applications.

The increasing importance of logic flaws is supported by other sources. See for example the OWASP Top 10 Security Risks from 2004 till 2013 [The04, The07b, The10, The13c], the Trustwave Global Security Report 2013 [Tru13], and the WhiteHat Website Security Statistics Report 2013 [Whi13]. These three sources mainly address special classes of logic vulnerabilities such as improper authentication, improper authorization, and information exposure vulnerabilities.

[1]We extracted the data by querying the CVE-DB with a set of known keywords such as authentication bypass or authorization bypass (we excluded bypasses caused by code injection vulnerabilities, e.g., SQL injection), and logic flaws. Then, we grouped the entries by year of disclosure. The number of flaws in web applications is calculated by refining the search criteria including the following keywords “php”, “asp”, “html”, “web”, “url”, and by excluding keywords such as “browser”, “firefox”, “chrome”, and “internet explorer”. We verified the quality of the data by classifying manually a sample of 10% of the population (we selected the sample randomly). About 75% of the sample is correctly classified.

OWASP Top 10 Security Risks 2013 ranked the improper authentication vulnerability as the 2nd most important risk in 2013 [The13c] overtaking Cross-Site Scripting vulnerabilities. Trustwave Global Security Report 2013 rated the overall logic flaw vulnerabilities 2nd in the top 10 application vulnerabilities 2013 [Tru13]. Moreover, it reported that 14 % of applications contain at least one logic flaw [Tru13]. Finally, WhiteHat Website Security Statistics Report 2013 reported that in 2012 information exposure vulnerabilities overtook Cross-Site Scripting vulnerability [Whi13] ranking it at the 1st position. The report estimated that in 2012 about 55 % of websites have at least one information exposure vulnerability [Whi13].

1.3 Objectives and Challenges

Researchers have proposed a number of techniques to detect vulnerabilities that can be used to test multi-party business applications. The choice of the technique depends on the information available to the tester including the software itself (e.g., the source codes) and the models describing the applications (e.g., formal specifications). However, developers of business application services do not share the source code with other organizations. As a result, source code-based testing techniques cannot be applied to multi-party business applications.

When specifications are available, the tester can use model checking techniques to explore the states of a formal model of the specifications to detect flaws in the logic. However, model checking offers no support to test real implementations. Finally, when both the source code and the specifications are not available, black-box testing techniques can be used. However, web scanners lack of the knowledge that is needed to detect logic flaws. This thesis aims at tackling these limitations. More specifically, the objectives of this thesis are the following:

Objective 1:

When a model is available, can we automatically verify whether

real systems expose a flaw discovered by model checking techniques?

To answer this question, we foresee several challenges. The main challenge are the translations between abstract elements of a model and the concrete world. The complexity of the translations depends from different factors, e.g., the choice of the testing interface, the relationship between model and real systems, and the vulnerabilities to discover.

First, a higher-level testing interface can remove too many details of the implementation and, as a result, the translation needs more powerful algorithms to reconstruct the missing information. On the contrary, a lower-level testing interface may carry too much information to the tester. As a result, the model may be too detailed making automatic reasoning unfeasible.

Second, the relationship between models and real system may not be necessarily 1-to-1. For example, the specifications of a standardized, security protocol are an informal model that describes a number of non-identical implementations. When testing different implementations, a rigid translation would prevent its reuse therefore making testing laborious.

Finally, the abstraction rules must take into account the type of vulnerability to be detected. For example, to test for cross-site scripting vulnerabilities, the tester sends malicious inputs inside an URL parameter. Then, the testing checks whether the response contains the same input. If the abstracting rules abstract away the response, then the tester may not be able to detect when the injection succeeded.

Objective 2

When models are not available, is it still possible to design an automated technique to detect flaws in the logic of multi-party business applications?

First, to detect logic flaws we need two types of model. The first type of model is a behavioral model of the implementation. The second model is a description of the logic that the application implements. These models can

be obtained by using model inference algorithms that learn a model with observations of the application.

The second challenge is related to the performance of the model-based testing techniques. In order to detect logic flaws, we need algorithms that are aware of the internal states of the application as well as of the application logic. These algorithms can be fully-fledged reasoning techniques such as model checking. However, most of the model-based testing techniques, model checking included, poorly scale because of the state explosion problem [FWA09], in which the state space to be explored could be sufficiently large to make it impracticable.

1.4 Contribution

The thesis makes the following contributions:

1. We present the design verification and testing via model checking of two authentication protocols. The analysis takes into account the different protocol flows as well as the protocol options. We show that given a formal model, the expected security properties, and a description of the implementations under test, we can execute the attacks discovered by the model checker against a number of implementations. Moreover, the testing techniques bridge the different abstraction layers in a generic and systematic way;

2. We show that when the specifications and the source code are not available, it is still possible to detect logic flaws following a black-box approach. Given a set of network traces and a description of the business function, our technique infers a model, generates test cases following an number of attack patterns, and then verifies whether the tests violated the business logic of the application;

3. We implemented our techniques into two prototypes. The prototypes demonstrated their effectiveness by discovering previously unknown logic flaws in real-size implementations;

4. We migrated the result of this thesis to SAP by (i) developing testing tools and (ii) supporting developers and engineers in assessing the security of the design and implementations.

1.5 Structure

The reminder of this manuscript is structured as follows.

Chapter 2 presents the related work to detect flaws for each of the following three categories: code analysis, model checking, and security testing. This section addresses the limitations and strengths of each category. First, this chapter shows that the source code analysis enables to detect input validation and application logic vulnerabilities. Second, it shows that model checking has been extensively used to detect logic flaws into security protocol specifications, web services, and business processes. Finally, it presents works in the area of security testing. This chapter presents also relevant work in the area of model inference.

Chapter 3 presents the two case studies of this thesis. The first case study are two web-based user authentication protocols whose specifications are publicly available, namely SAML Single-Sign On and OpenID protocols. The second case study is the eCommerce web applications. As opposed to the first case study, the specifications of web applications are never available in practice.

Chapter 4 presents the verification of the design of web-based user authentication protocols. This chapter makes the following contributions:

- it presents with great details the formal analysis via model checking of security protocols and their configuration options;
- it presents two logic flaws discovered by the model checking technique;
- it highlights that the verification of real implementations is still performed manually;

Chapter 5 addresses the first core-question. It tackles the problem of bridging design verification via model checking with the testing of real implementations. This chapter makes the following contributions:

- it presents a technique to bridge the gap between the two abstraction levels in a systematic way;
- it provides an interpretation of abstract attacks;
- it executes automatically attacks against five security protocol implementations;

Chapter 6 addresses the second core-question. It presents a novel technique that enables to test web applications when the specifications are not available. This chapter makes the following contribution:

- it presents a black-box passive model inference technique;
- it introduces an attack pattern-based test case generation algorithm;
- it demonstrates that real eCommerce web applications suffer from severe logic vulnerabilities

Chapter 7 presents the two migration activities performed in SAP. First, it shows an application of the results of Chapter 4 to the SAP implementations of SAML SSO. Second, it presents a tool implementing the technique of Chapter 5 to support SAP developers, engineers, or security analyst in verifying the design of protocols and to test them against putative attacks found by the model checker.

Chapter 8 draws the conclusions and gives an outline of the future work.

Chapter 2
Related Work

In this chapter, we review previous works describing techniques for detecting vulnerabilities in web applications, security protocols, and business processes. Furthermore, we present works in the area of model inference.

Researchers have proposed a number of techniques to detect vulnerabilities that can be used to test multi-party business applications. The choice of the technique depends on the information available to the tester. This information includes the software itself, in the form of source code or binaries, and explicit software documentation such as software specifications, and user manuals. According to the availability of this information, we can identify four testing scenarios as summarized in Figure 2.1. For each scenario, we associate the techniques that can be used.

In the scenarios at the top of Figure 2.1, the tester has access to the software. The tester can use different white-box security testing techniques such as static analysis, dynamic analysis, or taint analysis. At the bottom-left scenario of Figure 2.1, the tester has no information about the application. The tester can connect to it via the testing interface, and inspect the application by providing inputs and observing outputs. The inspection can be automated by using black-box security testing tools such as black-box web application scanners. Furthermore, black-box scanners can be combined with model inference techniques to obtain a model to guide the scanner. In the bottom-right scenario, the tester has the documentation of the application. Here, we assume that the documentation is sufficiently detailed to describe

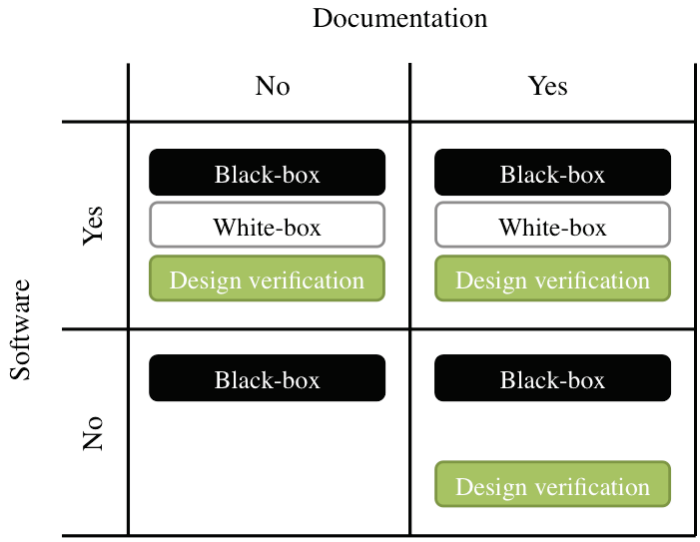


Figure 2.1: Testing Scenarios and Techniques

the behavior of the application. In this case, the tester can write a formal model and use design verification techniques such as model checking.

A number of causes limit the application of code analysis to multi-party business applications. First of all, software developers may not share the source code with the other developers. Second, the binaries of a service may be only available to the provider that executes them. Developers and providers may still perform the analysis of their own services. However, this leaves out of the scope vulnerabilities due to the composition, or caused by diverging security assumptions. For these reasons, in this thesis we will not consider white-box security testing techniques. However, to better present a complete picture of the security testing techniques, in this chapter we review also white-box techniques.

Structure: The remainder of this chapter is organized as follows. In Section 2.1, we present white-box security testing techniques. Then, in Section 2.2, we introduce the use of model checking for the design verification, and its application for supporting testing. Afterwards, in Section 2.3, we present black-box security testing techniques, and, in Section 2.4, we present

model inference algorithms and their applications. Finally, we conclude the chapter with a discussion in Section 2.5.

2.1 White-box Security Testing

In this section, we review relevant white-box testing techniques that can be applied when the source code of the application is available.

White-box testing tools combine together different code-based analysis techniques. The analysis begins by creating a model of the source code. The model can contain different aspects of a software program such as control flow, data flow, or both. Then, the model is analyzed by using different techniques. The choice of the technique mainly depends on the class of vulnerabilities to be detected.

To detect input validation vulnerabilities, white-box tools check whether the source code contains paths that allow untrusted input to reach databases or the output for the user. This can be done via model checking or taint-based analysis. Alternatively, the source code can be scanned to find SQL queries whose syntactical structure can be modified by the user inputs.

To detect logic flaws, white-box tools use model checking or custom algorithms looking for predefined patterns in the code. When a model checker is used, the security properties to be checked can be extracted in the form of code invariants via dynamic code analysis.

2.1.1 Detection of Input Validation Vulnerabilities

Huang et al [HYH+04a, HYH+04b] presented WebSSARI, a tool to discover injection vulnerabilities in PHP web applications. WebSSARI combines three techniques in one tool: static code analysis, lattice-based safety-type analysis, and bounded model checking. First, WebSSARI extracts a model with variable assignments, function calls, conditional structures, inputs and the outputs of the program. Second, it uses lattice-based safety-type analysis to assign a safety type to variables. Finally, the bounded model checker propagates the safety type of the input parameters to the variables

until the output is reached. The authors assessed WebSSARI against 230 open source web applications discovering 863 insecure statements. 607 of them were related to real vulnerabilities. Xie et al. [XA06] improved the approach of WebSSARI introducing a better support for the PHP language with inter-procedural analysis, dynamic typing, and conditional branches.

Livshits et al. [LL05] proposed a static analysis tool for detecting injection vulnerabilities in Java web applications. First, the user defines the vulnerability as a query in PQL [MLL05] (Program Query Language). A PQL query consists of the source Java objects (e.g. query string), the sink Java objects (e.g., database objects), and a set of rules to describe the data propagation between objects. Then, the tool performs a taint object propagation to derive sink objects starting from source objects. The authors assessed the tool on nine Java web applications searching for XSS, SQLi, and HTTP splitting vulnerabilities. The tool reported in total 41 vulnerabilities; 12 of them were false positives.

Jovanovic et al. [JKK10] presented Pixy, a tool based on static taint and data flow analysis of PHP source code. The tool analyzes the source code to identify vulnerable points. First, it propagates tainted input data (e.g., user inputs) into the code. Second, it checks if during the propagation the code contains special function calls to sanitization functions. Then, it checks whether the tainted data reached a database, or the user. In the former case, the tool discovered a SQLi, in the latter a XSS vulnerability. The tool analyzed seven PHP web applications of in total four million lines of code, discovering 409 injection vulnerabilities (213 XSS and 193 SQLi vulnerabilities) and raising 149 false alerts (89 XSS and 60 SQLi).

Wasserman et al. [WS07] proposed a technique to detect SQLi vulnerabilities in web applications. The approach takes in input a list of PHP files. First, it identifies input sources (e.g. URL parameters) and divides them into direct sources that provide data directly from the user, and indirect sources that provide data from other sources such as databases. Second, it performs a string and taint analysis of the query strings for inferring a Context-Free Grammar (CFG). The CFG is annotated to mark the parts of the grammar that are direct or indirect sources. Finally, it checks whether the language

generated by the CFG contains a command injection. The authors developed a tool and assessed it against five PHP web applications. The tool discovered 19 SQLi vulnerabilities and generated five false positives.

2.1.2 Detection of Application Logic Vulnerabilities

Balzarotti et al. [BCFV07] proposed MiMoSA (Multi-Module State Analyzer), a tool that detects workflow and data flows violations in PHP web applications. MiMoSA analyzes the source code in two phases. First, it builds a synthesis of the PHP files processing each file in isolation. For each of them, MiMoSA extracts a set of pre-conditions, post-conditions, sinks, and links to other PHP resources. The analysis of PHP code is done using the Pixy static analysis framework [JKK06]. During the second phase, MiMoSA infers the intended workflow and dataflow matching pre- and post-conditions of two PHP modules. Finally, MiMoSA uses a model checker to verify whether there are violations of the intended workflow and data flow. The authors tested the tool against five real-size PHP web applications detecting 32 vulnerabilities (six workflow and 26 data flow violations) with seven false positives.

Felmetsger et al.[FCKV10] proposed Waler, a tool for detecting logic flaws through the dynamic analysis of the source codes of J2EE-based web applications. First, Waler derives likely invariants of the Java servlets by using Daikon [EPG+07]. Then, it uses a modified version of the Java PathFinder model checker [NAS05] to detect violations of the invariants. The authors assessed Waler against 12 web applications, four of which are real-world applications. In total, Waler discovered 47 previously-unknown vulnerabilities and generated eight false positives.

Doupé et al. [DBKV11] presented a technique to detect a novel type of vulnerability called Execution After Redirect (EAR). EARs occur when the web application does not halt its execution after sending an HTTP redirection to the web browser. The authors developed a static source code analysis tool to detect EARs in Ruby on Rails web applications using heuristics to identify real vulnerabilities. The authors tested 18,127 web applications de-

tecting 3,944 EARs in 1,173 of them. 855 EARs were classified as exploitable because they caused unauthorized changes in the database, or information leakage to unauthorized users.

2.1.3 Discussion

As we have seen in the previous sections, white-box testing techniques have been used to discover both input validation vulnerabilities and logic flaws in web applications [BCFV07, FCKV10]. However, the source code of all the services of a multi-party business application are not available in practice. Therefore, these techniques cannot be applied to our scenario.

2.2 Design Verification

In this section, we review the relevant works in the area of design verification via model checking.

Model checking is a technique originally developed by Clarke et al. [CE82] and Quielle et al. [QS82]. It takes as input a model and a property and it explores the state-space of the model to verify whether the property is always satisfied. If the model does not satisfy the property, then the model checker produces a counterexample as a proof.

Two decades ago, Lowe proposed in a seminal work (See [Low96]) to use model checking to verify the design of security protocols. Afterwards, model checking has also been used to verify the design of web services and business processes. Furthermore, model checking has been used to analyze already deployed security protocols.

2.2.1 Design Verification

In this section, we first discuss the design verification of security protocols, starting from the seminal work of Gavin Lowe, and then covering similar applications to more complex security protocols. In the second part of the section, we discuss other applications of model checking to web services and business processes.

Security Protocols

Lowe was the first to propose the use of model checkers for the design verification of security protocols [Low96]. He applied model checking to verify the correctness of the Needham-Schroeder Public-Key (NSPK) authentication protocol [NS78] against an active attacker that is able to intercept, overhear, forge, and decompose messages (under the assumption that the attacker knows the cryptographic keys). Lowe modeled the honest protocol participants and the attacker in CSP (Concurrent Sequential Processes [Ros97]). Moreover, he modeled two parallel protocol executions. The first execution takes place between honest agents, while in the second, the attacker plays the role of a participant. The model checker discovered that the NSPK protocol is vulnerable to a man in the middle attack in which the attacker manages to be authenticated as one of the protocol participants.

Following this seminal work, model checking has been applied to other, more complex, security protocols. Donovan et al. [DNL99] reported the result of the verification of 51 protocols of the Clark-Jakob library [CJ97] by using the process algebra CSP [Ros97] and the model checker FDR (Failures-Divergence Refinement [FDR97]). The authors discovered that 16 protocols of the Clark-Jakob library are flawed.

Panti et al. [PST02] presented a formal analysis of the Kerberos authentication protocol [Ker00] by using NuSMV [CCGR99], a symbolic model checker. The authors discovered a vulnerability in Kerberos in which an attacker can intercept and reuse authorization tokens to create unrequested user sessions.

Mitchell et al. [MSS98] analyzed the Secure Socket Layer 3.0 Handshake Protocol [FKK11] with Murϕ [MMS97], a finite-state analysis tool. The authors performed an incremental analysis of the protocol that they named *rational reconstruction*. First, the authors started with a basic version of the protocol with essential message exchanges and omitting signatures and hashed data. Then, they added details to the protocol in an incremental way. At each step, they checked the correctness of the current version of the protocol with Murϕ, and if needed, they corrected adding new parts of SSL

3.0. At the end of this iterative process, they reached a model that resembles the SSL 3.0 protocol except for the parts explicitly omitted because of the perfect cryptography assumption. They authors discovered that a malicious client can force the server to switch to a lower and weaker version of the protocol.

Shmatikov et al. [SM99] and Armando et al. [ACC07] have analyzed the ASW protocol (Asokan, Shoup, and Waidner [ASW98]), a protocol to exchange contract signatures to allow the participants to reach mutual, non repudiable commitment on a previously agreed contract. Shmatikov et al. discovered two flaws in the protocol by using Murϕ [MMS97]. The first vulnerability lets the attacker replay messages of an old run of the protocol causing one of the participants to agree on an old version of a contract. The second vulnerability allows a malicious participant to cause agreement on inconsistent versions of the contract. Shmatikov et al. proposed a new version of the protocol fixing the flaws. Armando et al. [ACC07] analyzed the new version and discovered a further flaw. Armando et al. fed the SAT-based Model Checker [ACC07] (SATMC) with (i) a transition system modeling participants and the attacker, (ii) a set of LTL constraints modeling security properties of the communication channels, and (iii) LTL formulas as security properties. SATMC discovered an attack in which a malicious participant can obtain a different contract then the one on which the parties agreed.

Web Services and Business Processes

With the advent of Service-Oriented Architecture (SOA), researchers investigated the security issues of this new paradigm.

Salauüm et al. [SBS04] proposed the use of process algebra for modeling Web Services and verifying that their composition conforms to their requirements. The authors checked local properties of a single web service such as equivalences between processes, safety properties, and liveness properties. Moreover, they verified service choreography and orchestration for certifying compatibility. The authors used CWB-NC (Concurrency Workbench of the

New Century [CLS00]), a model checker for verifying finite-state concurrent systems. The model checker detected problems such as deadlocks and lack of synchronization.

Fu et al. [BFHS03, FBS04] analyzed different aspects of Web Services. First, they studied the relationship between the aggregated behavior of the composition of web services and the local behavior of single services [BFHS03]. Then, they extended their analysis to BPEL web services considering the message semantics [FBS04]. The authors proposed to translate BPEL web services into Promela [Hol04], and to use the SPIN model checker [Hol04]. They assessed the approach on a loan origination process using application-dependent LTL properties.

Backes et al. [BG05] discussed the risk of using the abstractions typical to the formal analysis. For example, the perfect cryptography assumption can leave undetected problems such as leaks due to length of encrypted data, and the abstraction of the time excludes timing attacks.

Backes et al. [BMPV06] presented a composed analysis of the Secure WS Reliable Messaging Scenario [DCV+05], a protocol that allows reliable message exchange between web services. First, the authors studied the protocol under the assumption of perfect cryptography by using the AVISPA tools [ABB+05]. Then, they manually analyzed the cryptographic primitives demonstrating classical cryptographic properties such as indistinguishability under adaptive chosen ciphertext attack (Ind-CCA2).

Schaad et al. [SLS06] presented an approach for the formal verification of delegation and revocation functionalities on the loan origination process in presence of static and dynamic separation of duty policies. The authors proposed to translate the workflow of the process from BPEL and ERP objects to NuSMV and then modeled the separation of duty policies as LTL constraints.

Wolter et al. [WMM09] presented an approach for the verification of access control security properties of business processes. The authors suggested to translate an augmented BPMN (Business Process Modeling Notation) with security annotation to Promela [Hol04]. Then, the model checker SPIN [Hol04] verifies the business process against a set of user-defined prop-

erties for detecting deadlocks.

Armando et al. [AP09] proposed an approach for modeling security-sensitive business processes with RBAC access control policies. They built a model checking problem where the model is a transition system translated from BPEL, and the property is any LTL formula. The authors discovered several flaws of the Loan Origination Process providing a corrected version. Arsac et al. [ACPP11] presented a similar approach in which the translation is from annotated BPMN models.

2.2.2 Model checking and Testing

In this section, we introduce works that discuss the use of model checking to generate test cases to test real implementations. We start with works that use model checking in latter phases of the life-cycle and, finally, we discuss the use of model checking when the model is not vulnerable.

Model Checking and Security Testing

Recently, model checking has been used to verify the security of already deployed security protocols. For example, Armando et al. [ACC+08] discovered a severe security flaw in the SAML-based Single Sign-On for Google Apps [Goo08]. The authors fed SATMC with (i) a transition system for the behavior of the participants and the attacker, (ii) a set of LTL constraints for modeling properties on the communication channels, and (iii) a LTL formula modeling the non-injective agreement property [Low97]. SATMC returned an attack in which a malicious SAML service provider can impersonate a legitimate user at any other service provider within the same federated environment. It is important to point out that the attack has been *manually* interpreted and reproduced against the implementation.

Guangdong et al. [GGJ+13] presented AUTHSCAN, a tool that combines a number of different techniques: model inference, static analysis of client-side script, model checking, and security testing. The first two techniques aim at inferring a model from a protocol implementation and will be detailed in Section 2.4. After the model inference step, the model is verified

by the model checker in the classical way. If the model checker returns a counterexample, AUTHSCAN translates it into a concrete test case. The translation replaces the abstract values of the counterexample with the real values learned during the model inference step. Furthermore, AUTHSCAN uses a user-defined test oracle to produce the test verdict after the test execution. The authors evaluated their tool against eight implementations of BroweserID (now called Persona [The13b]), Facebook Connect [Fac13], Windows Live ID [Mic13], and custom user authentication protocols. The authors reported that AUTHSCAN discovered seven security vulnerabilities of different type e.g. replay attacks, CSRF attacks, secret token leaks, and guessable tokens.

Model Checking and Mutation-based Security Testing

If the model checker does not find a counterexample in the model, it does not imply that the implementations are secure as well. In fact, implementations may be still vulnerable due to errors introduced by the developers. Dadeau et al. [DHK11] and later Büchler et al. [BOP11, BOP12a, BOP12b] proposed to apply the mutation-based testing technique to detect vulnerabilities.

Dadeau et al. proposed to mutate the model of a security protocol by injecting faults. Faults are injected by using the so called mutation operators. If the model checker finds an attack, the attack is used as a test case. Büchler et al. [BOP11, BOP12a, BOP12b] went beyond the prelimary work of Dadeau by proposing SPaCiTE, a tool for mutation-based, semi-automatic, security testing of web applications. The tool works as follows. First, the user selects the vulnerability to inject into the model. Then, the model checker verifies the security property on the mutation. If the model checker finds a counterexample, then it is concretized and executed against the real implementation. The concretization is done in two steps. First, the counterexample is mapped into concrete browser actions. To simplify the mapping, the authors proposed an intermediate language called WAAL (Web Application Abstract Language). Then, the counterexample is interpreted and executed against the web application. SPaCiTE has been assessed

against two lessons of WebGoat [The07a], an application vulnerable on purpose for educational purposes. The authors reported that they manage to detect successfully stored XSS, and lack of authorization vulnerabilities.

2.2.3 Discussion

In this section, we have presented works in the area of design verification via model checking. Model checking was applied to verify the design of security protocols, web services, and business processes, discovering previously unknown design flaws both during the design and at the deployment phases. Moreover, model checking has been proposed as a tool for supporting testing and security testing of real implementations.

However, these works showed the following shortcomings. First, the design verification via model checking focuses on the automatic detection of flaws in a model of the system under verification and falls short on testing the real system. In fact, the counterexamples returned by model checkers prove only that the model is flawed and it does not say whether the real system is also vulnerable. For example, a real system may solve the security flaw with additional and undocumented behaviors. As a result, in order to detect the flaw in implementations, the counterexamples are interpreted and reproduced against each implementation. To date, this activity is still done *manually*. Second, the mutation testing techniques have been assessed only on small applications to detect known vulnerabilities and there is still a lack of evidence of the scalability of these approaches to real systems. Moreover, the translations between models and real systems are specific to the web application domain, and they do not support cryptographic primitives, message composition and parsing.

In this section, we also presented AUTHSCAN whose authors claim it is able to automatically execute counterexamples against real implementations. It must be pointed out that the contribution of this thesis relatively to the execution of counterexamples is anterior to AUTHSCAN. Furthermore, the authors did not provide sufficient details about their techniques nor provided the tool. As a result, we could not perform any comparative analysis.

2.3 Black-box Security Testing

In this section, we present black-box security testing techniques.

This section is organized as follows. In Section 2.3.1, we present approaches for detecting input validation vulnerabilities. Then in Section 2.3.2, we introduce techniques for discovering application logic vulnerabilities.

2.3.1 Detection of Input Validation Vulnerabilities

To detect input validation vulnerabilities, we can use manual testing or (semi-)automatic black-box web application security scanners.

Black-box web application scanners are tool used for aiding the tester in detecting automatically or semi-automatically a wide spectrum of vulnerabilities. There are plenty of commercial and non-commercial web application scanners. A rich, yet incomplete, list is available at sectools.org [FF13].

The architecture of web application scanners is composed of three modules: the *crawler*, the *attacker* (or test vector set), and the *analysis* modules [BBGM10, DCV10]. The scan begins when the user provides an URL to the crawler. The crawler retrieves the page, extracts URLs from it, and requests the new pages. The crawler repeats this operations until a user-defined depth is reached, or until it does not find any new URL. The attacker module prepares and executes test cases in which it probes the web application with special inputs. The way the application is tested, and choice of the inputs depends from the type of vulnerability. Finally, the analysis module processes the pages in order to detect the vulnerability.

Bau et al. [BBGM10] and Doupé et al. [DCV10] presented two independent and, in a certain sense, complementary studies on black-box web application scanners. Bau et al. [BBGM10] studied the distribution of vulnerabilities into the wild correlating this distribution with the detection power of the scanners. Doupé et al. [DCV10] extensively benchmarked scanners against a wide range of vulnerabilities. In both works the authors performed controlled experiments on custom web applications to measure the coverage, the vulnerability detection rate, and the false positive rate.

The coverage of a web application measures the capability of the scanner of extracting URLs [BBGM10, DCV10]. URLs can be static strings in the HTML code, or generated dynamically by client-side scripts. Experimental results showed that scanners perform fairly well in discovering static URLs [BBGM10, DCV10]. Doupé et al. observed that the amount of surface the web application exposes to the scanner change dramatically with their internal state. Therefore, the coverage can be significantly improved by making scanners aware of the internal state [DCV10]. The experiments also showed that scanners perform poorly with dynamic URLs [BBGM10, DCV10]. However, the coverage slightly improves with text-based client-side scripts such as JavaScript and SilverLight. Bau et al. argue that this could be caused by the textual-based URLs extraction technique implemented by the scanners [BBGM10].

The detection rate measures the capability of the scanners in detecting vulnerabilities. Doupé et al. reported that 8 out of 16 vulnerabilities such as stored XSS/SQLi, logic flaws, and forceful browsing remained undetected [DCV10]. Similarly, Bau et al. also reported that SQLi vulnerabilities were not detected [BBGM10]. Moreover the detection rate of stored XSS, open redirects, HTTP response splitting, and flash parameter injection vulnerabilities was rather low [BBGM10]. Bau et al. attribute the low detection rate of advanced forms of XSS to the lack of a deep knowledge of the web application under test. For example, they reported that few scanners managed to inject a JavaScript code for a stored XSS but they failed in detecting the vulnerability. By correlating the capability of detecting single classes of vulnerabilities, Bau et al. observed that the testing emphasis for black-box web application scanners as a group is reasonably proportional to the verified vulnerability population in the wild [BBGM10].

The analysis of the false positives showed that the scanners with the highest detection rate are ranked among the one with the lowest number false positives [BBGM10]. Conversely, scanners with the lowest detection rate reported the highest false positive rate [BBGM10]. This indicates that false positives are rather a problem of the quality of the tools [BBGM10].

The experimental results of Bau et al. and Doupé et al. showed strengths

and limitations of scanners. First, the results on the coverage rates show room for improvement. Scanners have no notion of state and have a limited support for client-side scripts. This limits the capability of a scanner to crawl a web application. Second, the detection rates show that scanners focus on discovering the most common vulnerabilities such as reflected XSS. However, scanners have a low or null detection rate for a wide range of vulnerabilities such as stored XSS, SQLi, and logic vulnerabilities. Both authors attribute to the lack of knowledge of the state as the limitation in detecting certain classes of vulnerabilities.

Doupé et al. [DCKV12] presented a black-box state-aware vulnerability scanner, a tool containing a novel state-aware crawler and attacker module. The crawler module aims at inferring a Mealy machine [BJR08] by interacting with the web application. The details of the model inference technique are given in Section 2.4. During the inference process, the crawler uses the model for choosing the next link. It gives priority to the links that (i) do not cause a change of state, and (ii) are rarely or not yet visited. Once the inference phase is concluded, the attacker module takes as input the model for testing the application in a state-aware fashion. The test begins by resetting the state of the web application. Then it repeats the requests done by the crawler. If a request does not change the state of the application, the attacker module identifies inputs for probing the application with special values. Otherwise the module tries to explore the model looking for a path that brings to one of the previous state. If such a path does not exist, the attack module resets the application and repeats the same requests executed by the crawler. The authors did not developed a new fuzzing component, but rather integrated the one of the w3af tool [w3a13]. The authors run the tool against eight popular web applications comparing it with wget[SN12], w3af, and skipfish [the12]. In the experiments they considered three metrics, they are the number of discovered vulnerabilities, the number of false positives, and percentage of code coverage. The baseline for the code coverage is set to wget. The authors reported that their tool improved the code coverage of 66% in average from the baseline with a peak of 240%. w3af discovered six vulnerabilities and reported 10 false positives while their tool discovered nine

vulnerabilities and one false positive. The tool that discovered most vulnerabilities is skipfish that discovered 20 vulnerabilities (15 in the same web application) and seven false positives. However, their tool provided better code coverage than skipfish.

2.3.2 Detection of Application Logic Vulnerabilities

The OWASP Testing Guide v.3.0 [The08] suggests a 4-steps approach to test for application logic flaws in a black-box settings. First, the tester studies and understands the web application by playing with it and reading all the available documentation. Second, she prepares the information required to design the tests, including the *intended workflow* and the *data flow*. Then, she proceeds with the design of the test cases, e.g., by reordering steps or skip important operations. Finally, she sets up the testing environment by creating testing account, runs the tests, and verifies the results.

A number of prevalently manual methodologies have been recently proposed to detects more subtle vulnerabilities. Wang et al. [WCWQ11] presented a field study of the of Cashier-as-a-Service (CaaS) based web stores in which they developed a methodology that given a number of HTTP conversations of the same length, labels API arguments to show which ones an attacker could play with. The labeling rules are the following:

- A label can be of three types: S for the store, A for the attacker, and C for CaaS;
- Fresh values are labeled with the originators. For example, if the value of a argument **order_id** is originated by the store, then the argument is labeled with S;
- If a value is digitally signed, then it is labeled with the signing party. For example, if the argument **order_id** is signed by the store, then the argument is labeled with S.
- Any unsigned value is labeled with A;

- The labels are propagated to the subsequent arguments carrying the same value;

At the end of the labeling, the tester can design test cases by replacing the values of the arguments labeled with A. The test cases generation and tests execution are performed manually. Wang et al. applied their methodology and manually tested to two web stores, namely NopCommerce [nop13] and Interspire [Big13], discovering severe logic vulnerabilities in both software allowing an attacker to shop for free.

Wang et al. presented also a large-scale analysis of Single Sign-On protocols [WCW12] extending the earlier technique by enriching the labeling technique. First, the user collects three HTTP conversations between a web browser and the single sign-on protocol (SSO) implementation. Then, they label parameters similarly as described before. In this work they introduced new labels such as syntactic labels (e.g., decimal, boolean, and word), semantic labels (e.g., user-unique values and propagation chains), and read-write labels. Wang et al. applied this technique to real SSO implementations such as GoogleID (implementation of OpenID), Facebook Connect, JanRain, Freelancer.com, Nasdaq.com, and NYSenate.gov discovering 8 previously unknown flaws allowing the attacker to impersonate a victim at a relying party.

In Section 2.2.2, we presented the tool AUTHSCAN [GGJ+13] that detects logic flaws in SSO implementations. AUTHSCAN does not need the source code nor an initial model to generate and execute tests. This qualifies AUTHSCAN as a black-box security testing technique as the other approaches in this section.

2.3.3 Discussion

In this section, we have presented existing techniques to detect vulnerabilities in a black-box setting. Web scanners are tools used to explore the web application and then to prepare test cases to detect vulnerabilities. These tools perform well against specific classes of vulnerabilities such as reflected XSS. However, the lack of a model hampers the detection power of these tools such as stored XSS, stored SQLi, and logic vulnerabilities.

In this section we have seen that when black-box web application scanners are aware of the state of the application under test, then the coverage, and the detection power increase. However, these recent advances focused on input validation leaving the problem of detecting logic flaws in a black-box setting unexplored.

Recently, new ideas have been proposed to the black-box detection of logic flaws, which offer methodologies that highlight interesting parts of the data flow. However, these methodologies offer no support for the automatic generation and execution of tests. Moreover, automatic tools such as AUTHSCAN do not focus on vulnerabilities of the application logic of web application but on authentication issues of single sign-on protocols. Therefore, to date, the detection of logic flaws in web application is still done by manual inspection.

2.4 Model Inference

Model inference refers to a family of algorithms that derive a model from the observations of the behavior of an application. Model inference can be divided in two categories: *active* and *passive* learning. Active learning methods interact with the application under inference in order to explore its behavior whereas passive learning builds a model from a set of observations.

The remaining of this section is organized as follows. In Section 2.4.1, we present techniques and application of active learning techniques. In Section 2.4.2, we review passive learning techniques and their applications.

2.4.1 Active learning

We start off with the seminal paper by Dana Angluin. Angluin [Ang87] proposed the L* algorithm for learning an unknown regular languages L. The L* algorithm assumes that the alphabet of L is known and relies on an *oracle* to query on the membership of strings in L. The algorithm builds an internal table of strings called *observation table* representing the current knowledge of the algorithm about L. The table is updated by querying the

oracle about the membership of a string prepared by the algorithm. When the algorithm cannot decide whether there is a new state to explore and the table contains enough information to build a language, it proposes a conjecture S. Then, the oracle verifies S against L. If the two languages are the same, then the algorithm terminates. Otherwise, the oracle returns a counterexample that is a string w in the symmetric difference between L and S. The algorithm extends the table with w and provides a new conjecture. The L* algorithm has been proposed to infer a DFA from the implementation by using the implementation as the oracle [PVY01], and experiments as membership queries.

Hossen et al. [HGR11] proposed to use the L* algorithm to infer a DFA of web application for security testing. However, the algorithm cannot be applied directly to web applications for the following reasons. First, web applications accept parametric inputs (i.e. URL with query strings and/or POST data). Second, web applications generate dynamically output messages whose contain part of the inputs for the next communications. As a result, the assumption that the input alphabet is a priori known is no longer valid. Hossen et al. [HGOR13] proposed to solve the first issue by modeling web application as Extended Finite State Machines (EFSM). Then, they proposed to use a state-aware crawler for discovering the inputs before the inference begins [HGOR13].

Doupé et al. [DCKV12] proposed to learn a model of the application while crawling the web application. We already described the testing technique in Section 2.3.2 and in this section, we detail the model inference technique. The inference algorithm is based on three sub-algorithms for page clustering, state change detection, and state clustering. The authors proposed to model web applications as Mealy machines [BJR08]. A Mealy machine is an automaton whose the output is determined by the current state and the input. The input symbols are the URLs including the query string and POST data. The output symbols are abstraction of HTML pages. A single web page is represented as a prefix tree of vectors for anchors and forms. Each vector contains the DOM path, the URL domain and path, a list of parameters, and the value of the parameters. A group of pages can be represented in a

similar way. All the prefix trees are transformed into a vector such that the i-th element of a page vector contains the list of nodes of the tree at depth i. These vectors are stored in a further prefix tree called *Abstract Page Tree* (APT) whose leaves are HTML page. The page clustering algorithm visits the APT seeking for internal nodes that satisfy certain properties based on the number of leaves and depth of the subtrees. The pages belonging to the subtree rooted in the chosen node are clustered together.

2.4.2 Passive learning

Li et al. [LX11] proposed BLOCK, a black-box tool that learns model by observing HTTP conversations to block attacks. BLOCK infers a model of the web application and a set of invariants on the session variables. BLOCK models a web application as a stateless machine that receives an input and returns an output. The input of the application is the URL, the parameters and the session variables. The output is a synthesis of a web page and the session variables. Web pages are clustered in four steps. The clustering technique borrows the first 2 steps from TEXT [KS11]. First, the web pages are transformed in a set of DOM paths. Second, the list of paths are pruned in order to keep only the essential paths of the page. This is done by calculating the number of pages in the conversation that contain a path. Third, pages are clustered by similarity. Two pages are similar if they have similar essential paths. Finally, each page in the HTTP conversation is compared against the template it belongs to and the essential paths are removed. The remaining paths are the output parameters. The second part of the training mode consists in calculating three different types of invariants, they are: invariants between inputs, between an input and an output, and between consecutive input/output pairs. Invariants are calculating with Daikon [EPG+07]. The resulting model and invariants are used by BLOCK to intercept and block attacks to the web application.

In Section 2.2.2, we presented AUTHSCAN, a black-box tool that use a number of techniques to test SSO implementations. In this section, we provide the details on its model inference phase. The model inference algorithm

takes in input HTTP conversations and extract an initial model in TML (Target Model Language [WL93]). The model is refined then with static code analysis of the JavaScript code. In this step, AUTHSCAN looks for known function calls for arithmetic operations, cryptographic operations, and concatenations. Furthermore, AUTHSCAN uses a differential fuzzing analysis for removing redundant data by probing the SSO implementation with mutated data in order to identify differences in responses. AUTHSCAN performs also a type inference on data to identify strings, integers, and booleans.

Dury et al. [DHP09] described an approach for passively learning a model of web-based business applications. The authors used Parameterized Finite Automaton (PFA) for modeling the applications that enriches the classic notion of finite automaton [HMU06] with guards on transitions and parameters on states. PFA models both control flow and data flow of an application. The authors consider each input traces as a PFA, then they merge all the PFAs into a single PFA that in turn is abstracted into a symbolic PFA. Dury et al. proposed to use data mining algorithms like C4.5 [Qui93] to infer guards of a symbolic PFA from a PFA. The final model is then translated into Promela [Hol04] and passed to SPIN [Hol04] for verifying application-dependent properties.

2.4.3 Discussion

In this section, we have presented applications of model inference techniques to security testing. We have seen that both active and passive techniques have been used for infering models for different purposes. Model inference has been used to detect input validation vulnerabilities, to detect attacks against web applications, to detect flaws specific to the single sign-on domain, and to test application-dependent properties. To best of our knowledge, there are no applications of model inference to obtain a model to be used for the generation of test cases to detect logic flaws.

2.5 Conclusions

In this chapter, we have reviewed existing techniques to detect vulnerabilities. The choice of the testing technique depends on the availability of the software source code and documentation.

We have seen that when the source code is available there is great variety of techniques that can be used. However, in multi-party business application the source code is not available in practice. Therefore, in this thesis we do not consider white-box testing techniques.

When the specifications are available, we have seen that design verification via model checking is quite effective in detecting logic flaws. However, model checking falls short when verifying already deployed protocols and the counterexample proves the existence of a vulnerability at model-level. As a result, counterexamples are often interpreted and executed manually against real implementations.

When even models are not available, automated tools such as black-box web scanners can be used. Black-box security testing tools are very effective in detecting vulnerabilities such as XSS and SQLi. However, these tools are not capable of detecting vulnerabilities in the logic of the application. As a result, logic flaws are still detected by manual inspection.

Chapter 3
Case Studies

In this thesis, we use two case studies to show how the testing techniques we propose can be applied to real world scenarios. The fundamental aspect that differentiates these two case studies is the availability of the specifications.

The first case study describes a web-based single sign-on protocol used by multi-party business applications. In this thesis, we consider the SAML 2.0 Web-based Single Sing-On and the OpenID authentication protocols whose specifications are publicly available. The design of these protocols is verified in Chapter 4 via model checking, while their real implementations are tested in Chapter 5.

The second case study is an eCommerce web application. eCommerce web applications are multi-party business applications whose specifications are not available. In this thesis, we use this case study in Chapter 6 to assess the black-box testing technique we propose to detect logic flaws without a starting model.

Structure: This chapter is organized as follows. In Section 3.1, we introduce the SAML 2.0 Web-based Single Sing-On and the OpenID authentication protocols. Then, in Section 3.2 we introduce the eCommerce web applications.

3.1 Case Study 1: Web-based Single Sign-On Protocols

The OASIS Security Assertion Markup Language 2.0 [OAS08] Web browser Single Sign-On (hereafter SAML SSO) and the OpenID Authentication Protocol [Ope07] (hereafter OpenID) are two emerging standards that enable partners of multi-party business applications to authenticate their users once, and then access the services of the application without the need to be authenticated again. SAML SSO and OpenID implementations are part of identity management software such as SAP NetWeaver Identity Manager and IBM Tivoli Federated Identity Manager, as well as by online services such as the Google Apps suite (e.g., GMail and Google Calendar). Everyday, millions of users are authenticated by using these two protocols. For example, Google claims that more than 5 millions of companies use SAML and OpenID to login their users at Google Apps[1].

3.1.1 The SAML 2.0 Web browser Single Sign-On

SAML SSO is a standardized, open, and interoperable authentication protocol. In this respect, it offers a significant number of configuration options allowing it to be applicable in a multitude of environments. It is based on an XML format for encoding security assertions as well as a number of protocols and bindings that prescribe how assertions should be exchanged in a variety of applications and/or deployment scenarios. Three 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 authenticates C and issues corresponding authentication assertions (a special type of assertions used to authenticate users). The SSO protocol ends when SP consumes the assertions generated by IdP to grant or deny C access to the requested resource.

A SAML SSO profile offers two main usages depending on whether the

[1] See http://www.google.com/enterprise/apps/business/

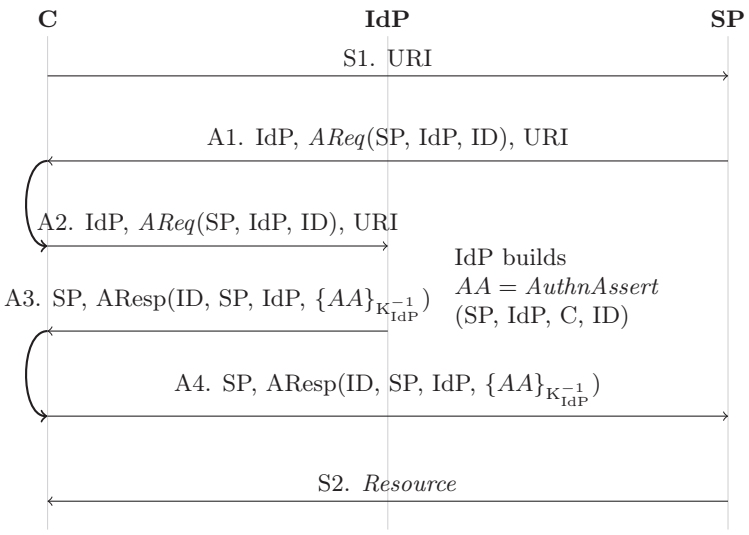


Figure 3.1: SAML SSO SP-initiated with front channels

web user requests a resource from an SP by contacting the SP directly (SP-initiated SSO), or by contacting the IdP that presents a set of SP resources that web users can consume (IdP-initiated SSO). Both SP-initiated and IdP-initiated SSO can be used in combination with the *artifact resolution protocol* (ARP) that provides a mechanism by which SAML messages can be transported by reference instead of by value. In addition, SAML SSO offers many configuration options ranging from optional fields in messages, usage of SSL/TLS at transport layer, encryption, digital signature, etc.

In the rest of this section we detail both SP-initiated and IdP-initiated SAML SSO variants with and without ARP. The use of the ARP is often referred to as *back channel*, while *front channel* indicates that the artifact resolution protocol is not used. In this chapter we use the latter naming.

SAML SSO SP-initiated with front channels

Figure 3.1 shows the reference flow for the SAML SSO SP-initiated variant with front channels. In step S1, C asks SP to provide the resource located at the address URI. SP then initiates the protocol by sending C a redirect response (e.g., HTTP 302 Response message) directed to IdP con-

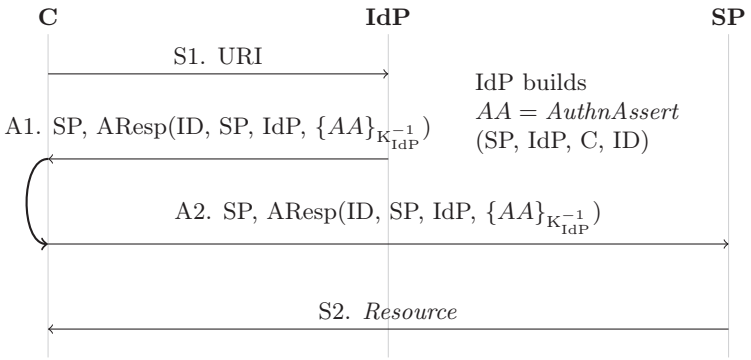


Figure 3.2: SAML SSO IdP-initiated with front channels

taining an authentication request of the form $AReq(\mathrm{SP}, \mathrm{IdP}, \mathrm{ID})$ where ID is a string uniquely identifying the request. IdP then challenges C to provide valid credentials and if the authentication succeeds IdP builds an authentication assertion $AA = AuthnAssert(\mathrm{SP}, \mathrm{IdP}, \mathrm{C}, \mathrm{ID})$ and places it into a response message $\mathrm{AResp}(\mathrm{ID}, \mathrm{SP}, \mathrm{IdP}, \{AA\}_{\mathrm{K}_{\mathrm{IdP}}^{-1}})$, where $\{AA\}_{\mathrm{K}_{\mathrm{IdP}}^{-1}}$ is the assertion digitally signed with $\mathrm{K}_{\mathrm{IdP}}^{-1}$, the private key of IdP. SAML does not prescribe how the IdP authenticates C. This is thus abstracted away from our formalization. In our analysis we assume that a successful user authentication takes place.

IdP then places AResp into an HTML form and sends it back to C (SAML POST Binding). The response is forwarded by using a client-side script that triggers the POST submission to SP. This completes the message exchange and SP can deliver the requested resource to C.

SAML SSO IdP-initiated with front channels

The message flow is shown in Figure 3.2. As opposed to SP-initiated, C asks IdP to access SP's resources (step S1 in Figure 3.2). Once C authenticates with IdP, IdP initiates the SAML Authentication Protocol by issuing an authentication assertion. The execution of the protocol continues as seen in Figure 3.1.

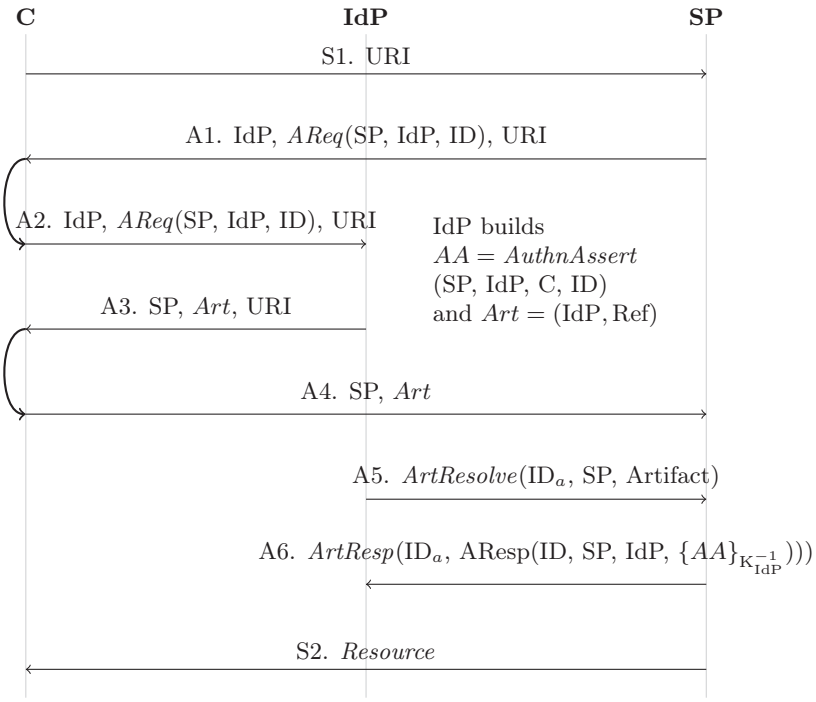


Figure 3.3: SAML SSO SP-initiated with back channels

Back channels (Artifact Resolution)

In Figure 3.1 and Figure 3.2 SAML messages are directly exchanged by using C as intermediary. The SAML profiles exposing SAML messages to the web browsers are called front channel profiles. In addition, SAML defines another method for exchanging SAML messages. Instead of relaying through the web browser SAML messages, SPs and IdPs exchange references called *artifacts*. Then, they run the ARP to resolve artifacts in SAML messages.

The ARP can be used for exchanging SAML requests as well as SAML responses. Figure 3.3 shows an SP-initiated SSO in which back channels are used to resolve only the SAML response.

The protocol flow for exchanging the SAML request is the same as seen in Figure 3.1. After having authenticated C, in step A3 the IdP prepares the authentication assertion AA. Additionally, it prepares the artifact $Art =$

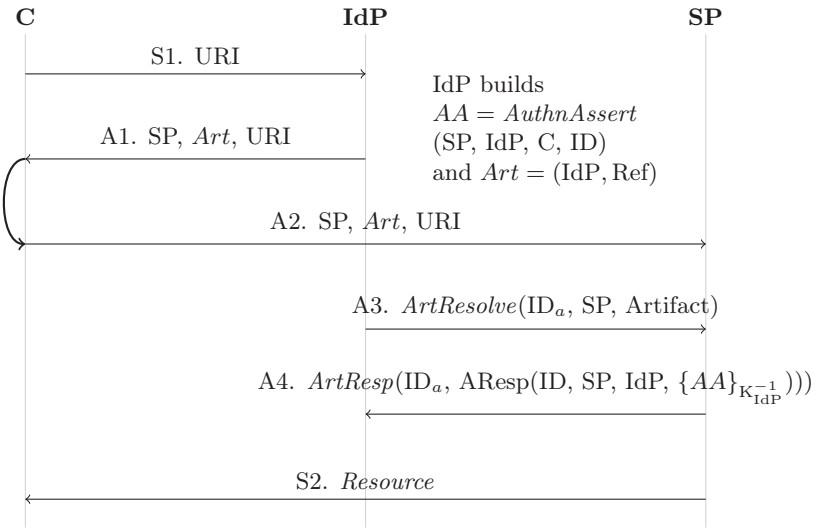


Figure 3.4: SAML SSO IdP-initiated with back channels

(IdP, Ref) associated to AA. Then IdP stores in a database the AA and the Art. Finally IdP sends Art to C that in turn forwards to SP (step A4). Upon receiving Art, SP establish a direct channel with IdP and sends the ArtifactResolve(ID_a, SP, Art) where ID_a is a string uniquely identifying the resolve request (step A5). Then, IdP fetches the assertion associated to the artifact, encapsulates the SAML response into a ArtifactResponse(ID_a, ArtifactResolve(...)), and finally sends it back to SP. The protocol ends when SP serves the resource to C.

Similarly as seen for the SP-initiated profile, the IdP-initiated profile uses back channels. Figure 3.4 shows the IdP-initiated profile with back channels.

Security Assumptions

The security of the SAML SSO protocol relies on a number of assumptions about the trustworthiness of the principals involved as well as the security of the transport protocols employed.

Protocol Participants Concerning trustworthiness, the protocol assumes that:

A1 : IdP is trustworthy for both C and SP;

A2 : SP is not trustworthy.

Secure Transport Layer The SAML 2.0 specifications repeatedly state the following assumptions of the transport protocols used to carry the protocol messages:

TP1 : Communication between C and SP can be carried over a unilateral SSL/TLS channel, established through the exchange of a valid certificate (from SP to C).

TP2 : Communication between C and IdP is carried over a unilateral SSL/TLS channel that becomes bilateral once C authenticates itself on IdP. This is established through the exchange of a valid certificate (from IdP to C) and of valid credentials (from C to IdP).

An analysis of the SAML specifications reveals that the standard does not specify whether the messages exchanged at steps S1 and A4 of Figure 3.1 and of Figure 3.3 must be transported over the same SSL/TLS connection or whether two different SSL/TLS connections can be used for this purpose. In other words, there is a certain degree of ambiguity on how assumption *TP1* of Section 3.1.1 can be interpreted.

The reuse of the SSL/TLS connection at step S1 to also transport the message at step A4 is at first sight the most natural option. However this is difficult to achieve in practice for a number of reasons:

Resuming SSL/TLS connections The use of a single SSL/TLS connections for the exchange of different messages cannot be guaranteed as, e.g., the underlying TCP connection might be terminated (e.g. timeout, explicitly by one of the end points), an SSL server could not resume a previously established session, or a client might be using a browser that very frequently renegotiates its SSL connection.[2]

[2]See, for instance, `http://publib.boulder.ibm.com/infocenter/tivihelp/v2r1/index.jsp?topic=/com.ibm.itame2.doc_5.1/am51_webseal_guide54.htm`

Software modularity Nowadays, software is designed to be increasingly modularized, capitalizing on layering and separation of concerns. This may result in the fact that—within SP implementations—the software module that handles SAML messages has no access to the internal information of the transport module that handles SSL/TLS. Thus, the information on whether the client has used a single SSL/TLS connection or two different ones may not be available.

Distributed SPs The SAML SP may be distributed over multiple machines, e.g., for work-balancing reasons. This results in physically different SSL/TLS endpoints, with the inherent impossibility of enforcing a single session for all communications between SP and C.

3.1.2 The OpenID Authentication Protocol

OpenID is an open and user-centric web browser-based single sign-on protocol. It provides a way to authenticate a user C by asking her to prove that she controls a valid user identifier [Ope07]. OpenID is decentralized in the sense that it does not require relying parties (SPs) and OpenID identity providers (IdPs) to have a pre-established relationship. It also does not rely on an existing infrastructure on which a central authority approves or registers relying parties or OpenID providers. The OpenID Authentication 2.0 specification [Ope07] describes an *authentication protocol* and an *association session protocol*. It also prescribes how messages are transported over HTTP messages defining two communication types: direct communication and indirect communication. The former is established between service and identity providers, the latter involves the user agent as intermediary. In the next section, we describe in more details the different protocols.

Authentication Protocol

The protocol is initiated by C who access a resource URI at SP providing SP an identifier that C has to prove to control. The identifier is used by SP to identify which IdP C uses for authentication. Then C is redirected to IdP together with an authentication request. Once C proves to control the

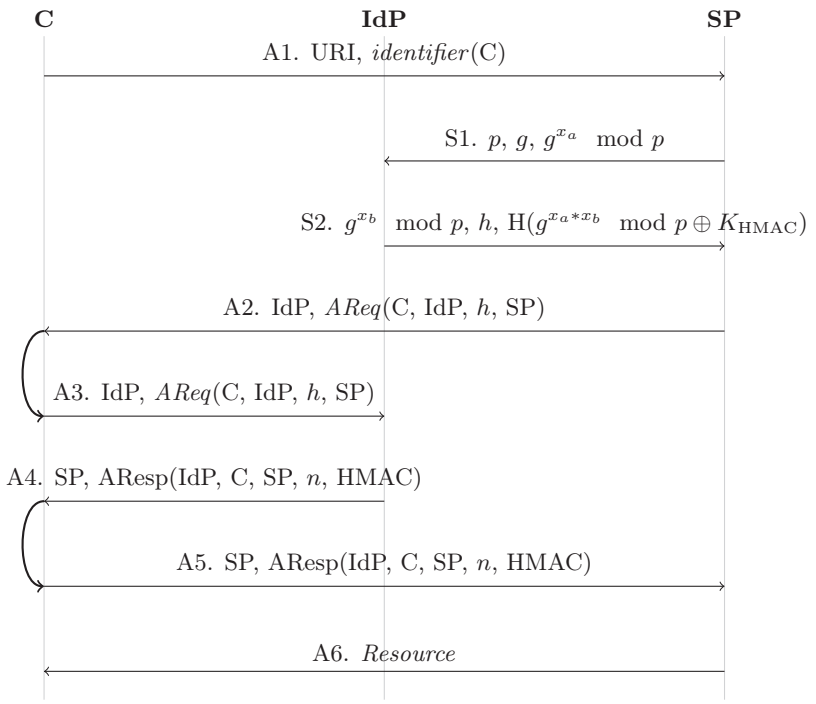


Figure 3.5: OpenID authentication protocol with Diffie-Hellman session association

identifier, IdP issues and signs a positive assertion and redirects C back to the SP transporting the response. SP checks the validity of the signature and it lets C access resources available at its site. The manner in which C is challenged is out of the scope of the protocol specification.

Association Session Protocol

The association session protocol establishes a shared secret K_{HMAC} between SP and IdP used to sign and verify authentication responses. SP initializes this protocol by sending an association session request to IdP right after SP discovers which IdP C uses to authenticate. IdP returns to SP a shared secret together with a value h called *handle* used as a key to refer to associations. OpenID specifies only two ways to transmit K_{HMAC}, that are: *No-Encryption* association sessions and *Diffie-Hellman* (D-H) association

session. When No-Encryption is used the IdP sends a response with K_{HMAC} in plain-text, whereas when D-H association is deployed a D-H shared key is calculated in order to encrypt K_{HMAC}. No-Encryption is used only over a secure transport layer.

OpenID Authentication Protocol with Session Association

Figure 3.5 depicts the authentication protocol flow used in combination with the D-H session association. In step A1, C sends to SP his identifier `identifier(C)`. SP identifies IdP using C's identifier (this procedure is not considered here, we just assume that SP has a look-up table) and then initiates the D-H association session protocol. At the end of its execution, SP receives an handle for the association h and a shared secret K_{HMAC} (step S2); then SP issues an authentication request in step A2. C, acting as intermediary, redirects the request to IdP (step A3), which challenges C and issues an assertion within an authentication response accordingly. The information sent to SP is signed calculating an HMAC over IdP, C, SP, a nonce n and the handle h (step A4). In step A5, C delivers the response to SP. If SP accepts the response, then it will send a resource back to C.

Security Assumptions

Protocol Participants OpenID works under the assumptions that IdP is not compromised and that IdP is trusted by SPs to generate authentication assertions. The latter requires a certain care from SPs as in principle any entity can claim itself to be an IdP. SPs are assumed to be capable to select those IdPs that can be considered trustworthy.

Secure Transport Layer The OpenID specifications strongly recommend the use of SSL connections for all parts of the interaction, including communication with the user. Not following this recommendation would make the OpenID protocols vulnerable in many trivial aspects that may not fit relevant business scenarios. In our analysis we follow this recommendation and we assume that the protocol is working under the assumptions TP1 and

TP2 as discussed in Figure 3.1.1.

3.1.3 Security Goals

In this section, we introduce the security goals of web-based single sign-on protocols.

By comparison with multiple credentials web authentication schemes (one username-password pair per service), it is natural to expect that at the end of the execution the protocol fulfills the following mutual authentication goal: SP authenticates C, and C authenticates SP.

Furthermore, when SP uses SSL/TLS to send the resource to C, we also expect that the protocol offers confidentiality of the resource, i.e., the resource will remain a secret between C and SP.

3.2 Case Study 2: eCommerce Applications

The term eCommerce refers to the activities of buying and selling goods and services over electronic communication systems. eCommerce covers a wide range of forms of commerce including business-to-consumer commerce (e.g., online stores, marketplace, video streaming platforms, auction systems, and online gambling), business-to-business commerce (e.g., procurement, producer-wholesaler or wholesaler-retailer transactions, and so on), and business-to-government commerce. Modern eCommerce is carried on over the web, and it can be accessed via web applications, typical for online stores, or via electronic data interchange (EDI) services for the exchanged of commercial data between business partners. In this thesis, we focus on online stores for business-to-consumer commerce accessible via web applications.

eCommerce web applications are software solutions for buying and selling products or services over the web between online stores and consumers. eCommerce applications implement on-line catalogues and virtual shopping carts in which customers place the items they would like to buy. An example of a workflow of a eCommerce web application is shown in Figure 3.6. eCommerce web applications have a front-end for the customers, and a back-

Figure 3.6: The workflow of eCommerce web applications

office interface for the store administrators and employees. Consumers place orders by using the front-end of the store. While in the back-office of the store, clerks process the orders by fetching the items from the warehouse and preparing them for the delivery. Finally, the store ships the items and an invoice to the customer.

eCommerce web applications integrate the payment into the purchase process. This is done by using the API provided by online payment services such as PayPal, Amazon Payments, Google Checkout, or Authorize.NET[3]. The integration can be done at different steps of the purchase process and it depends on the payment system that is integrated. In Figure 3.7, we show an example of integration that requires two HTTP redirections. In step S1, U adds the item I into the cart. The store confirms the operation showing the updated cart in step S2. In step S3, U confirms the order. Then, S redirects U to P together with the details of the cart such as the name of

[3]For a rich, yet incomplete, list of payment systems see `https://en.wikipedia.org/wiki/List_of_online_payment_service_providers`.

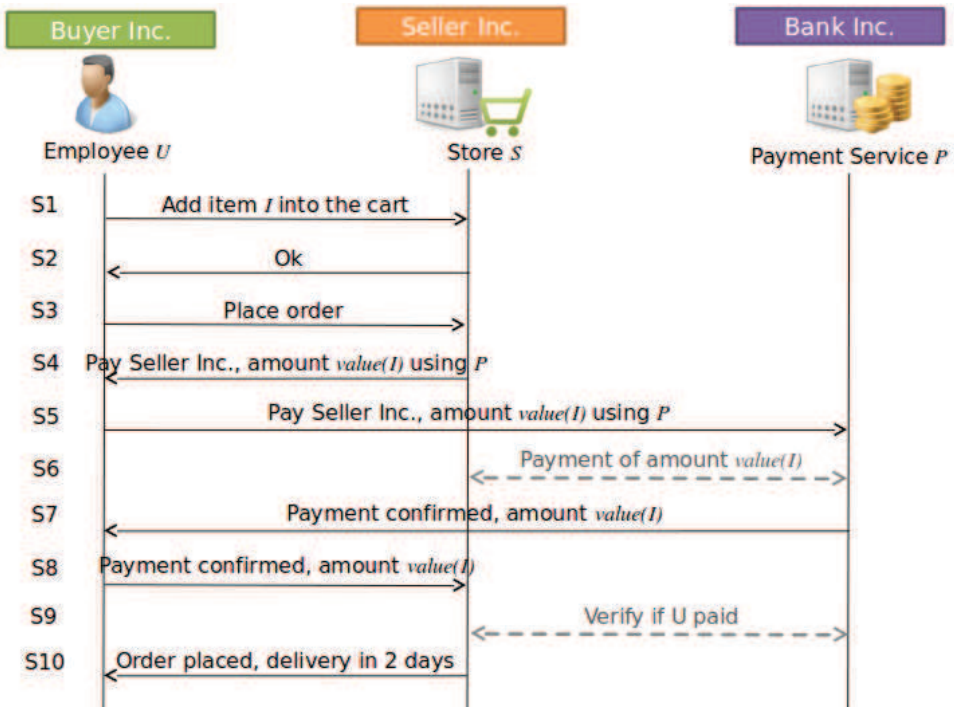


Figure 3.7: eCommerce web application and payment systems

the payee, the amount of the monetary transaction, and so on. In step S6, P may notify S about the result of the money transfer authorized by U. This step is optional and it depends on the API provided by the service P. In step S7, U has authorized the payment for the amount of I and is redirected back to the store. In step S8, the store may query P for checking the result of the transaction. The purchase ends with a confirmation of the purchase. The steps S6 and S9 could be performed offline by the clerk when processing the order.

3.2.1 Application Logic

At the end of the purchasing process, the parties involved have the following expectations:

1. the buyer and the seller agreed on the goods of the purchase;

2. the buyer, the seller, and the bank agreed on the amount of money to transfer from the buyer to the seller;

3. the bank transferred the amount of money from the account of the buyer to the one of the seller;

4. the seller delivers the goods to the buyer.

Therefore, we expect that the software fulfills the above requirements.

3.3 Conclusions

In this chapter we introduced two case studies. The first case study consist of two web-based authentication protocols: SAML SSO and OpenID. On this two protocols, we will apply a model checking technique for detecting flaws in their design (See Chapter 4). Then, we will address the question of whether also real implementations suffer from the flaws identified by the model checker (See Chapter 5). The second case study is a typical eCommerce web application for which no formal specifications are available. In this case, we will apply a model inference technique in order to extract a sufficiently expressive model to perform model-based security testing in a black-box scenario (See Chapter 6).

Chapter 4
Model Checking of Web-based Authentication Protocols

When the software specifications are available, it is possible to use automated reasoning techniques such as model checking to detect vulnerabilities in applications. In this section we present a novel application of model checking for the security analysis of authentication protocols. Our study led to the discovery of a previously unknown logic flaw into the design of SAML SSO and OpenID. By exploiting this vulnerability, an attacker can hijack a client authentication attempt or force the latter to access a resource without its consent or intention. In this section, we also discuss the manual tests required to verify the presence of the design flaw in actual protocol implementations. We tested three SAML SSO implementations and two OpenID implementations discovering that four out of five are vulnerable to the logic flaw discovered by the model checker. Moreover, we discovered that the design flaw can be exploited as a launching pad of Cross-Site Scripting attacks in the SAML-base SSO for Google Apps. All our findings have been discussed with members of the OASIS Security Services Technical Committee and a SAML V2.0 Errata has been redacted and approved [OAS12].

Structure: This chapter is organized as follows. Section 4.1 presents the formal models of SAML SSO and OpenID. Section 4.2 introduces the protocol options, the formal analysis, and the results. Section 4.3 presents the

authentication flaw we discovered in SAML SSO and OpenID. Then, Section 4.4 discusses the manual tests against real implementations. Finally, Section 4.5 draws the conclusions.

4.1 Formalization

This section presents the formalization of SAML SSO and OpenID protocols. This section focuses on four main aspects of the model: messages, behavior of participants, protocol sessions, and the security goal. For each participant we differentiate the behavior according to the profile.

The specification language that we use in this section is ASLan++ (the AVANTSSAR Specification Language [vOM11]), one of the specification language of the AVANTSSAR platform [AAA+12]. However, when we analyzed SAML SSO, the ASLan++ language was still under development and we used the HLPSL++, another specification language of the AVANTSSAR platform.

4.1.1 AVANTSSAR Specification Language

In this section we introduce the main concept of ASLan++[1]. ASLan++ is a formal language for specifying service-oriented architectures, security policies, and security properties. An ASLan++ specification is a set of encapsulated *entities*. Entities may model web services as processes and their static compositions. In our case studies, we model the client, the service provider and the identity provider as entities. The top-level entity is usually called *Environment*, and it is used to "glue" together the inner entities. The definition of an entity contains a list of parameters, the symbol section, and the body section. The list of parameters represent what the entity knows at the beginning of its execution. The symbol section contains the declaration of types, variables, constants, functions, macros, and algebraic equations that are accessible only from the entity and its inner entities. The body of an entity contains the logic. For example, an entity modeling a web service

[1] A detailed explanation, as well as tutorials and software are available at `http://www.avantssar.eu`.

performs message sending and message receiving actions while a composition of entities instantiates two entities with appropriate parameters. ASLan++ supports common programming language-like statements such as if-then-else for deterministic conditional branches, select-do for non deterministic conditional branching, while loops, as well as message passing primitives such as message sending and receiving statements.

The security goals can be expressed in temporal logic [Pnu77, Hol04] or state formulas, while the policies are expressed as Horn clauses [Hor51].

4.1.2 Formalization of SAML SSO

In this section we present the formal model of the SAML SSO protocols shown in Figure 3.1, Figure 3.2, Figure 3.3 and Figure 3.4.

Structure of a Specification

The SAML SSO specification is structured as follows:

```
1  entity Environment {
2    symbols
3      % Protocol Message
4
5    entity Session ( ... ) {
6
7      entity Client ( ... ) {
8        % [...]
9      }
10     entity ServiceProvider ( ... ) {
11       % [...]
12     }
13     entity IdentityProvider ( ... ) {
14       % [...]
15     }
16     body {
17       % Instantiation of a single SAML SSO protocol run
18     }
19
20   } goals:
21     % G1 and G2
22   body {
23     % Instantiation of several SAML SSO protcol run
24   }
```

```
25 }
```

The innermost entities are the `Client` entity, `ServiceProvider` entity, and `IdentityProvider` entity. These entities are contained in the `Session` entity that model a single protocol run by instantiating the principal and providing the appropriate parameters. The outermost entity is `Environment` that instantiates parallel protocol runs.

Protocol Messages

We model messages, their structure, encapsulation, message encoding and fields by using ASLan++ function symbols and constants. We declare them in the symbol section of the `Environmant` entity as follows:

```
1  symbols
2    %% HTTP protocol values
3    get, post                                                  : method;
4    code_30x, code_200                                         : code;
5    uri_sp, uri_i                                              : uri;
6    c, sp, idp                                                 : agent;
7    id                                                         : int;
8    %[...]
9
10   %% HTTP Messages
11   httpReq(method, agent, http_element, http_element          : message;
12   httpResp(code, agent, http_element, http_element)          : message;
13   %% HTML elements
14   htmlForm(agent, saml_binding)                              : http_body;
15   %% SAML Messages
16   aReq(agent, agent, int)                                    : saml_message;
17   noninvertible
18       signedAResp(private_key, agent,
19                           agent,agent, int)                  : saml_message;
20   %% SAML bindings
21   hBind(saml_message, uri)                                   : saml_binding;
22   pBind(saml_message, uri)                                   : saml_binding;
23
24   % [...]
```

Symbols for function such as `httpReq`, `httpResp`, `aReq` and `authnResponse` model single protocol messages. `hBind` and `pBind` represent message bindings. Constants `get`, `post`, `code_200` and `code_30x` model the HTTP GET method,

the HTTP POST method, HTTP 200 response code, and the HTTP 30x-family response codes.

We express protocol message encapsulation by composing together symbols. For example, the authentication request `aReq(sp,idp,id)` transported over an HTTP message is expressed as follows:

```
httpResp(code_30x, idp,
         hBind(aReq(sp, idp, id), uri),
         nil_http_element)
```

Communication Channels

ASLan++ supports three different abstractions for communication channels, they are Abstract Channel Model (ACM), Cryptographic Channel Model (CCM), and Ideal Channel Model (ICM). In our models, we used the ACM model. As opposed to the other channel models, ACM explicitly refers to communication channels by using element of the formal language. We believe that this feature better suites to the purpose of this thesis that is testing real implementations. Nevertheless, the other channel models could be used as well, however the testing technique introduced in Chapter 5 does not support them yet.

ACM refers to channel by ASLan++ constants of the type `channel`. An agent `A` sends a message `M` to `B` over the channel `ch` by using the following ASLan++ primitive:

```
1 A -ch-> B: M;
```

While `A` receives a message `M` from `B` over the channel `ch` by using the following statement:

```
1 B -ch-> A: M;
```

In the following section use the following naming convention for channels. We use the variable name `Ch_X2Y` of type `channel` for representing a communication channel between the agent X and the agent Y in which X is the sender and Y is the intended recipient. Moreover, we assume that every time two agents communicate they use a new pair of channels.

Client Entity

The `Client` entity takes four parameters called `Actor`, `SP`, `IdP` and `URI` representing, respectively, the agent that plays the entity of `Client`, the service provider, the identity provider, and the resource that C wants to access.

```
1  entity Client (Actor, SP, IdP: agent,
2                 URI : agent,
3                 % [...]
4                 ) {
5    symbols
6      AReq : hBind(aReq (agent, agent, int), agent);
7      ARsp : pBind(signedAResp (inv(public_key),
8                               agent, agent, agent, int), uri);
9          % [...]
10 }
```

C is a web browser guided by a user. We model C as a standard browser unaware of protocols encapsulated in HTTP messages. We model this behavior by using ASLan++ compound types[2]. The ASLan++ code above shows the declaration of two compound types `Areq` and `ARsp`, respectively, the SAML authentication request and response. We use compound type also for modeling SAML artifact messages.

SP-initiated with front channels As shown in Figure 3.1, a client participating to the SAML SSO SP-initiated profile with front channel performs the following actions. First, C initiates its run by sending an HTTP request for a resource and receiving an authentication request over a 30x HTTP response (step S1 and A2). Second, C executes the 30x redirection type and receives from the IdP and authentication response within an HTML form (step A2 and A3). Finally, C sends the authentication response to the SP and receives the resource.

The `Client` entity for the SAML SSO SP-initiated with front channels takes in input 5 parameters more, they are `Ch_C2SP_1`, `Ch_SP2C_1`, `Ch_C2IdP`, `Ch_IdP2C`, `Channels`, respectively for the communication channel for request-

[2]The same aspect could have been captured by using the most general ASLan++ message type. However, compound types have the characteristic that they prune the research space of the model checker.

ing a resource to SP, for receiving the authentication request from SP, for forwarding the authentication request to IdP, for receiving the authentication response from IdP, and a set of pairs of channels for forwarding the authentication response to SP. We will clarify the use of the set of channels later.

The `Client` entity is the following:

```
1  entity Client ( %[...]
2                  Ch_C2SP_1, Ch_SP2C_1,
3                  Ch_C2IdP, Ch_IdP2C: channel,
4                  Channels: agent.channel.channel set) {
5    %% Compound types
6    % [...]
7    body {
8      %% S1-A1
9      Actor -Ch_C2SP_1-> SP : httpReq(get, URI, nil_http_element,
10                                     nil_http_element);
11     SP -Ch_SP2C_1-> Actor : httpResp(code_30x, IdP, ?AReq,
12                                     nil_http_element);
13
14     %% A2-A3
15     Actor -Ch_C2IdP->  IdP : httpReq (get, IdP, AReq, nil_http_element
           );
16     IdP -Ch_IdP2C-> Actor  : httpResp(code_200 , nil_agent,
17                                      nil_http_element, htmlForm(?AnySP,
                                           ?ARsp));
18     if(Channels->contains((?AnySP, ?Ch_C2SP_2, ?Ch_SP2C_2))) {
19       %% A4-S2
20       Actor -Ch_C2SP_2-> AnySP : httpReq (post, AnySP,
21                                            nil_http_element, ARsp);
22       AnySP -Ch_SP2C_2-> Actor : httpResp(code_200, nil_agent,
23                                            nil_http_element, ?
                                                 Resource);
24     }
25   }
26 }
```

Here, the `Client` entity uses `AnySP` for fetching the new channels `Ch_C2SP_2` and `Ch_SP2C_2` respectively for sending the `ARsp` and receiving the `Resource`.

IdP-initiated with front channels The `Client` entity of SAML SSO IdP-initiated with frond channels does not request the resource to SP. Instead, it requests the resource hosted by SP to the IdP and receives the

authentication response from the IdP (step S1 and A1 of Figure 3.2). The `Client` entity's body begins as follows:

```
1   body {
2     %% S1-A1
3     Actor -Ch_C2IdP-> IdP : httpReq (get, URI, nil_http_elements,
4                                          nil_http_element);
5     IdP -Ch_IdP2C-> Actor : httpResp(code_200 , nil_agent,
6                                  nil_http_element, htmlForm(?AnySP, ?
                                       ARsp));
7     %% A2-S2 == A4-S2 of SP-initiated with front channels
8     %[...]
9   }
```

The steps A2-S2 of the IdP-initiated profile with front channels are the same of the steps A4-S2 of the SP-initiated profile with front channels.

Back channels When back channels are used, SP and IdP do not exchange SAML messages through C. Instead, they use C for redirecting references. This requires to modify the structure of the messages received by the `Client` entity. This is done in two points. First, in the symbol section and then in the body. For example, let us consider the SP-initiated with back channels of Figure 3.3. The `Client` entity is derived from the entity of SAML SSO SP-initiated with front channel as follows:

```
1   entity Client (Actor, SP, IdP: agent,
2                  URI : agent,
3                  % [...]
4                  ) {
5     symbols
6       ARes : hBind(aReq (agent, agent, int), agent);
7       Art  : artBind(artifact(agent, int), agent);
8
9   body {
10      %% S1-A1
11      % [...]
12
13      %% A2-A3
14      Actor -Ch_C2IdP-> IdP: httpReq (get, IdP, AReq, nil_http_element);
15      IdP -Ch_IdP2C-> Actor: httpResp(code_200 , nil_agent,
16                                      nil_http_element, htmlForm(?AnySP,
                                           ?Art));
17      if(Channels->contains((?AnySP, ?Ch_C2SP_2, ?Ch_SP2C_2))) {
18        %% A4-S2
```

```
19        Actor -Ch_C2SP_2-> AnySP: httpReq (post, AnySP,
20                                            nil_http_element, Art);
21        AnySP -Ch_SP2C_2-> Actor: httpResp(code_200, nil_agent,
22                                            nil_http_element, ?
                                                Resource);
23      }
24    }
25  }
```

ServiceProvider entity

The general `ServiceProvider` entity has four parameters. They are `Actor`, `IdP`, `C` and `URI` representing the agent that plays the entity of `ServiceProvider`, the identity provider, the client, and the resource it hosts. The other parameters are the channels whose number depends from the specific profile.

```
1  entity ServiceProvider(Actor, IdP, C: agent,
2                         URI : uri,
3                         % [...]
4                         ) {
5    % [...]
6  }
```

SP-initiated with front channels In the SP-initiated profile with front channels, the service provider, upon receiving a request for a resource, initiates the protocol by issuing an authentication request (See steps S1-A1 of Figure 3.1). Then, it waits for incoming authentication responses, and finally it returns the requested resource (steps A4-S2 of Figure 3.1).

The `ServiceProvider` entity of the SP-initiated profile with front channels takes four parameters more for the communication channels, they are `Ch_C2SP_1`, `Ch_SP2C_1`, `Ch_C2SP_2`, and `Ch_SP2C_2`.

```
1  entity ServiceProvider(% [...]
2               Ch_C2SP_1, Ch_SP2C_1,
3               Ch_C2SP_2, Ch_SP2C_2: channel
4               ) {
5    % [...]
6    body {
7      %% A1-S1
8      C -Ch_C2SP_1-> Actor: httpReq(get, URI, nil_http_element,
9                                    nil_http_element);
```

```
10      ID := fresh();
11      Actor -Ch_SP2C_1-> C: httpResp(code_30x, IdP,
12                                     hBind(aReq(Actor, IdP, ID),
13                                                        URI),
                                                            nil_http_element
                                                            );
14
15      %% A4-S2
16      C -Ch_C2SP_2-> Actor: httpReq (post, Actor, nil_http_element,
17                                  pBind(signedAResp(inv(pk(IdP)),
18                                                        Actor, IdP, C, ID),
                                                            URI));
19      Resource := fresh();
20      Actor -Ch_SP2C_2-> C : httpResp(code_200, nil_agent,
21                                     nil_http_element, Resource);
22    }
23  }
```

`fresh` is a reserved ASLan++ keyword for generating nonces. The `ServiceProvider` uses it for both generating the unique ID of the authentication request `ID` and for modeling the resource `Resource`.

SP-initiated with back channels When back channels are used the service provider executes the artifact resolution protocol. In Figure 3.2, SP executes the ARP in step A5 and A6. We model the service provider as follows:

```
1   entity ServiceProvider(% [...]
2               Ch_C2SP_1, Ch_SP2C_1,
3               Ch_C2SP_2, Ch_SP2C_2,
4               Ch_SP2IdP, Ch_IdP2SP: channel) {
5     % [...]
6     body {
7       %% S1-A2 of the SP-initiated with front channels
8
9       %% A4
10      C  -Ch_C2SP_2-> Actor: httpReq (get, Actor, nil_http_element,
11                                      artBind(artifact(IdP, ?Ref), URI);
12      %% A5
13      Actor -Ch_SP2IdP -> IdP: httpReq(get, IdP, nil_http_element,
14                                      artResolve(IDa, SP, artifact(IdP,Ref
                                            )));
15      %% A6
16      IdP -Ch_IdP2SP -> Actor: httpResp(code_200, nil_agent,
```

```
17                                  nil_http_element,
18                                  aReq(Actor, IdP, ID));
19      %% S2
20      Resource := fresh();
21      Actor -Ch_SP2C_2-> C: httpResp(code_200, nil_agent,
22                                  nil_http_element, Resource);
23    }
24  }
```

IdP-initiated with front channels In the IdP-initiated profile, the `ServiceProvider` entity waits for any incoming authentication response that matches `IdP`, `Actor`, and it is signed with `inv(pk(IdP)` key own by the identity provider (step A1 of Figure 3.2) . Then, it returns the resource (step S2) We model this as follows:

```
1  entity ServiceProvider(% [...]
2                 Ch_C2SP_2, Ch_SP2C_2: channel) {
3    % [...]
4    body {
5      %% A2-S2
6      C  -Ch_C2SP_2-> Actor: httpReq(post, Actor, nil_http_element,
7                                pBind(signedAResp(inv(pk(IdP)),
8                                                  Actor, IdP, C, ?ID), URI)
                                                      );
9      Resource := fresh();
10     Actor -Ch_SP2C_2-> C: httpResp(code_200, nil_agent,
11                                nil_http_element, Resource);
12   }
13 }
```

IdP-initiated with back channels The `ServiceProvider` entity of the IdP-initiated profile with back channels is the following:

```
1  entity ServiceProvider(% [...]
2               Ch_C2SP_1, Ch_SP2C_1,
3               Ch_C2SP_2, Ch_SP2C_2,
4               Ch_SP2IdP, Ch_IdP2SP: channel) {
5    % [...]
6    body {
7      %% A2
8      C  -Ch_C2SP_2-> Actor: httpReq(get, Actor, nil_http_element,
9                                artBind(artifact(IdP, ?Ref), URI);
10     %% A3
```

```
11      Actor -Ch_SP2IdP -> IdP: httpReq(get, IdP, nil_http_element,
12                                       artResolve(IDa, SP,
13                                              artifact(IdP, Ref)));
14      %% A4
15      IdP -Ch_IdP2SP -> Actor: httpResp(code_200, nil_agent,
16                                  nil_http_element,
17                                  signedAResp(inv(pk(IdP)), Actor, IdP,
18                                         C, ID));
19      %% S1
20      Resource := fresh();
21      Actor -Ch_SP2C_2-> C: httpResp(code_200, nil_agent,
22                                  nil_http_element, Resource);
23    }
24  }
```

where A3 and A4 are the execution of the ARP.

IdentityProvider entity

The general `IdentityProvider` entity takes four parameters: `Actor`, `C`, `SP` and `TrustedSPs` representing, respectively, the agent that will play the identity provider entity, the client, the service provider, and a set of trusted service providers. The general `IdentityProvider` entity is the following:

```
1  entity IdentityProvider (Actor, C, SP: agent,
2          TrustedSPs: agent set,
3          % [...]
4          ) {
5      % [...]
6  }
```

SP-initiated with front channels The IdP of the SP-initiated profile with front channels performs three steps. First, it waits for incoming authentication request from `C`. Second, it checks if the issuer of the authentication request is a trusted service provider. Finally, it sends back to the client an authentication response. This is modeled as follows:

```
1    body {
2      %% A2
3      C -Ch_C2IdP-> Actor: httpReq (get, Actor,
4                                  hBind(aReq(?SP, Actor, ?ID),
5                                  ?URI), nil_http_element);
6      if (TrustedSPs->contains(SP)) {
```

```
7         %% A3
8         Actor -Ch_IdP2C-> C: httpResp(code_200, nil_agent,
9                                 nil_http_element,
10                                htmlForm(SP, pBind(
11                                signedAResp(inv(pk(Actor)), SP,
12                                       Actor, C, ID), URI)));
13      }
14    }
```

SP-initiated with back channels When back channels are used, the identity provider does not send back the authentication response. Instead, first it stores the response locally and the service provide associates a reference to the request. Second, it sends the reference to the client. Finally, it waits for executing the ARP (steps A5-A6).

```
1     body {
2       %% A2
3       C -Ch_C2IdP-> Actor: httpReq (get, Actor,
4                                hBind(aReq(?SP, Actor, ?ID),
5                                ?URI), nil_http_element);
6       if (TrustedSPs->contains(SP)) {
7         Ref := fresh();
8         DB->add(Ref, signedAResp(inv(pk(Actor)), SP, Actor, C, ID));
9         %% A3
10        Actor -Ch_IdP2C-> C: httpResp(code_200, nil_agent,
11                                      nil_http_element,
12                                      arpBind(artifact(Actor, Ref), URI
                                            ));
13        %% A5
14        SP -Ch_SP2IdP-> Actor: httpReq(get, Actor, nil_http_element,
15                                      artResolve(?IDa, SP,
16                                         artifact(Actor, ?AnyRef)));
17        if (DB->contains(?AnyRef, signedAResp(inv(pk(?AnyIdP)),
18                    ?AnySP, ?AnyIdP, ?AnyC, ?AnyID))) {
19            %% A6
20            Actor -Ch_IdP2SP -> SP: httpResp(code_200, nil_agent,
21                                        nil_http_element,
22                                        signedAResp(inv(pk(AnyIdP)),
23                                              AnySP, AnyIdP, AnyC,
                                                    AnyID));
24        }
25      }
26    }
```

IdP-initiated with front channels The identity provider of the IdP-initiated profile wait for a request for a resource from the client (step S1). Then it sends an authentication response to the client (step A1). We modeled it as follows:

```
1   body {
2   %% S1
3   C -Ch_C2IdP-> Actor: httpReq(get, URI, nil_http_element,
4                                      nil_http_element);
5   if (TrustedSPs->contains(SP)) {
6       %% A1
7       Actor -Ch_IdP2C-> C: httpResp(code_200, nil_agent,
            nil_http_element,
8                                  htmlForm(SP, pBind(
9                                  signedAResp(inv(pk(Actor)), SP,
10                                   Actor, C, ID), URI)));
11    }
12  }
```

IdP-initiated with back channels The entity of the identity provider of the IdP-initiated profile with back channels is the following:

```
1   body {
2     %A1
3     C -Ch_C2IdP-> Actor: httpReq(get, URI,
4                               nil_http_element, nil_http_element);
5     if (TrustedSPs->contains(SP)) {
6       Ref := fresh();
7       DB->add(Ref, signedAResp(inv(pk(Actor)), SP, Actor, C, ID));
8       %% A2
9       Actor -Ch_IdP2C-> C: httpResp(code_200, nil_agent,
10                               nil_http_element,
11                               arpBind(artifact(Actor, Ref), URI));
12      %% A3
13      SP -Ch_SP2IdP-> Actor: httpReq(get, Actor, nil_http_element,
14                                 artResolve(?IDa, SP,
15                                    artifact(Actor, ?AnyRef)));
16      if (TrustedSPs->contains(?AnyRef,
17                                 signedAResp(inv(pk(?AnyIdP)),
18                                 ?AnySP, ?AnyIdP, ?AnyC, ?AnyID))) {
19          %% A4
20          IdP -Ch_IdP2SP -> Actor: httpResp(code_200, nil_agent,
21                                       nil_http_element,
22                                       signedAResp(inv(pk(AnyIdP)),
```

```
23                                                AnySP, AnyIdP, AnyC, AnyID))
                                                      ;
24          }
25        }
26      }
```

Sessions and Environment

The scenarios that we considered involve a trusted identity provider, two service provides, one of which is malicious, and a client. In our analysis we suppose that a user asks for two different resources (e.g. by using two instances of a web browser) being authenticated by the same identity provider. We specified this scenario in the `Environment` entity that instantiates two protocol executions. A single protocol execution is defined in the `Session` entity that in turns instantiates participants.

The fragment below shows the body of the `Environment` and `Session` entities for the SAML SSO SP-initiated with front channels:

```
1  entity Environment {
2    % [...]
3    entity Session (C, IdP, SP: agent,
4                    TrustedSPs : agent set,
5                    URI : agent,
6                    Ch_C2SP_1, Ch_SP2C_1, Ch_C2SP_2, Ch_SP2C_2,
7                    Ch_C2IdP, Ch_IdP2C: channel,
8                    Channels: agent.channel.channel set) {
9
10     entity Client (
11       % [...]
12     } entity IdentityProvider (
13       % [...]
14     } entity ServiceProvider (
15       % [...]
16     } body {
17       %% New protocol run
18       new Client(C, SP, IdP, URI,
19                  Ch_C2SP_1, Ch_SP2C_1,
20                  Ch_C2IdP, Ch_IdP2C,
21                  Channels);
22       new ServiceProvider(SP, IdP, C, URI,
23                           Ch_C2SP_1, Ch_SP2C_1,
24                           Ch_C2SP_2, Ch_SP2C_2);
25       new IdentityProvider(IdP, C, SP, TrustedSPs,
```

```
26                              Ch_C2IdP, Ch_IdP2C);
27      } goals
28          %% Mutual authentication goal
29          G1a:(_) C *->  SP;
30          G1b:(_) SP *-> C;
31          %% Confidentiality goal
32          G2:( _) {C, SP};
33
34    }
35    body {
36      TrustedSPs := {sp, i};
37      CChannels := {(sp, ch_c2sp_2s1, ch_sp2c_2s1),
38                    (i,  ch_c2i_2s2,  ch_i2c_2s2)};
39      %% Two protocol runs
40      % honest agents
41      new Session(c, idp, sp,
42                  TrustedSPs,
43                  uri_sp,
44                  ch_c2sp_1s1, ch_sp2c_1s1, ch_c2sp_2s1, ch_sp2c_2s1,
45                  ch_c2idp_s1, ch_idp2c_s1,
46                  CChannels);
47      % malicious sp (==i)
48      new Session(c, idp, i ,
49                  TrustedSPs,
50                  uri_i,
51                  ch_c2i_1s2, ch_i2c_1s2, ch_c2i_2s2, ch_i2c_2s2,
52                  ch_c2idp_s2, ch_idp2c_s2,
53                  CChannels);
54    }
55  }
```

Security goals

As we said in Chapter 3, we expect that SAML SSO fulfills the mutual authentication property G1 and the confidentiality of the resource G2.

We specify the G1 and G2 security properties in the `goals` section of the `Session` entity. This is modeled by using ASLan++ labels. The mutual authentication is modeled with two labels defined in the goal section of the `Session` entity: `G1a:(_)C *-> SP` and `G1a:(_)SP *-> C`. The labels `G1a G1b` are then used to mark the data value upon which the agents `SP` and `C` agree on. For example, when C sends the message S1 to SP, the variable URI is marked as follows: `URI`. Similarly, we define the confidentiality property:

`G2( _){C, SP}`. In this case, the property states that only `C` and `SP` shares a data value marked by `G2`.

4.1.3 Formalization of OpenID

In this section we present the formal model of OpenID in Figure 3.5.

In this subsection we present a formalization in ASLan++ of OpenID. It abstracts away the steps of the protocol considered irrelevant for the analysis we perform in this case study, more precisely, the IdP discovery phase, and the association session.

Structure of a Specification

An OpenID formal specification is structured as follows:

```
26
27  entity Environment {
28    symbols
29      % Protocol Message
30
31    entity Session ( % Parameters
32                     ) {
33
34      entity Client ( % Parameters
35                     ) {
36        %[...]
37      }
38      entity ServiceProvider ( % Parameters
39                     ) {
40        %[...]
41      }
42      entity IdentityProvider ( % Parameters
43                     ) {
44        %[...]
45      }
46      body {
47        %Instantiation of a single OpenID protocol run
48      }
49
50    } goals:
51      % G1 and G2
52    body {
53      % Instantiation of several OpenID protocol run
```

```
54   }
55 }
```

Protocol Messages

We model messages, their structure, encapsulation, message encoding and fields by using ASLan++ function symbols and constants as follows:

Listing 4.1: Model excerpt.

```
1
2  symbols
3    %% HTTP protocol values
4    get, post           : method;
5    code_30x, code_200 : code;
6    uri_sp, uri_i       : uri;
7
8    %% HTTP Messages
9    httpReq(method, agent, http_element, http_element)   : message ;
10   httpResp(code, agent, http_element, http_element)    : message ;
11   htmlForm(agent, http_element)  : http_body;
12
13   %% OpenID Messages
14   aReq( agent, agent, int, agent)                      :
         oid_authn_message;
15   aResp(agent, agent, agent, int, int, hmac)           :
         oid_authn_message;
16   noninvertible
17     hmac(symmetric_key, agent, agent, agent, int, int) : hmac;
18
19 }
```

The function symbols `aReq` and `aResp` model the structure of a protocol message. The function symbols `httpReq` and `httpResp` describe the structure of a HTTP message that can be used to transport the previous OpenID messages. Constants `get`, `post`, `code_200` and `code_30x` model the HTTP GET method, the HTTP POST method, HTTP 200 response code, and the HTTP 30x-family response codes. The function `hmac` models the structure of the shared secret between `SP` and `IdP`.

Communication Channels

We used the ACM model and the same naming convention as seen for SAML SSO.

Client Entity

The `Client` entity takes nine parameters. The first three are the agents involved in a protocol run `Actor`, `IdP`, `SP`, respectively for the agent that will play the entity of `Client`, the OpenID provider, and finally, the service provider. The parameter `URI` is the address of the resource the end-user would like to access. The further parameters in the `Client` entity declaration refer to the ACM channel pairs used for exchanging messages with the other protocol entities.

Listing 4.2: Client entity.

```
1  entity Client (Actor, IdP, SP: agent, URI : uri,
2                 Ch_C2SP_1, Ch_SP2C_1, Ch_C2IdP, Ch_IdP2C: channel,
3                 Channels: agent.channel.channel set) {
4    symbols
5      AnySP       : agent;
6      Resource    : http_element;
7      AReq        : aReq( agent, agent, int, agent);
8      AResp       : aResp( agent, agent, agent, int, int, hmac);
9      Ch_C2SP_2   : channel;
10     Ch_SP2C_2   : channel;
11
12     % [...]
13         }
```

The symbol section in `Client` introduces variables used locally in the body. The next ASLan++ code fragment will make clear in which context they are used:

Listing 4.3: Sample interaction.

```
1  body {
2    %% A1-A2
3    Actor -Ch_C2SP_1-> SP    : httpReq (post, URI, nil_http_element,
4                                         identifier(Actor));
5    SP    -Ch_SP2C_1-> Actor : httpResp(code_30x, ?IdP, ?AReq,
       nil_http_element);
```

```
6     %% A3-A4
7     Actor -Ch_C2IdP->   IdP    : httpReq(get, IdP, AReq,
          nil_http_element);
8     select {
9       on(IdP -Ch_IdP2C-> Actor : httpResp(code_30x, ?AnySP, ?AResp,
10                                                  nil_http_element) &
11          Channels->contains((?AnySP, ?Ch_C2SP_2, ?Ch_SP2C_2))): {
12                %% A5-A6
13            Actor -Ch_C2SP_2-> AnySP: httpReq(get, AnySP, AResp,
                  nil_http_element);
14            AnySP -Ch_SP2C_2-> Actor: httpResp(code_200, nil_agent,
15                                                nil_http_element, ?
                                                      Resource);
16      }
17    }
18  }
```

The first step in the protocol is performed by C who accesses a resource `URI` at SP providing it with an identifier, `identifier(Actor)`, that C has to prove to control. The corresponding HTTP response redirects the user agent towards `IdP`, given the HTTP 30x code contained in it. The HTTP response carries also the authentication request `AReq`, which contains information about the client that is requesting to initialize the protocol.

In the next step, C proves control of the identifier to `IdP`. Upon the reception of a positive response from `IdP`, `C` is redirected back to `SP` transporting the `AResp` response.

It is important to remark that the configured channels are being checked to verify the relationship between the service provider and this client (as we will discuss later). Finally, the contacted `SP` will provide `C` with a `httpRespCookie` containing the requested resource.

OpenID Provider Entity

The `IdentityProvider` entity takes as parameters, respectively, the agents participating to the session `Actor`, `SP`, and `C`. Then, the parameter `Shared_key` represents the shared secret key used to sign and to verify assertions. Finally, the parameter `Handle` is a value used as a pointer to refer to the `Shared_key`. `SP` is supposed to hold valid credentials in order to issue an authentication requests to `IdP`. Additional parameters are the communication channels

used in the body of the entity. The variable `Nonce` is a fresh value sent in the redirection of `C` to `SP`. Note that the actual authentication of `C` towards `IdP` is abstracted away.

Listing 4.4: IdentityProvider model.

```
entity IdentityProvider (Actor, SP, C: agent,
                         Shared_key: symmetric_key,
                         Handle: int) {
  symbols
    Nonce: int;
  body {
    select{
      on(C  *->* Actor: httpReq(get, Actor, aReq(C, Actor, Handle, SP)
          ,
                            nil_http_element)): {
        Nonce := fresh();
        Actor *->* C: httpResp(code_30x, SP, aResp(Actor, C, SP, Nonce
            , Handle,
                            hmac(Shared_key, Actor, C, SP, Nonce,
                                Handle)),
                           nil_http_element);
      }
   }
  }
}
```

Service Provider Entity

This entity takes the same agent parameters as the identity provider plus some specific set parameters such as: `Discovery`, used to abstract the discovery sub-protocol; `ConsumedNonces`, initially empty, used to check the freshness of the `aResp`. Other parameters are `Shared_key` and `Handle`, already explained before. The remaining parameters consist in the communicating channels used here.

Listing 4.5: ServiceProvider entity.

```
entity ServiceProvider (Actor, IdP, C: agent, URI : uri,
                        Discovery: agent.agent set,
                        ConsumedNonces: int set,
                        Shared_key: symmetric_key,
                        Handle: int,
```

```
6                          Ch_C2SP_1, Ch_SP2C_1, Ch_C2SP_2, Ch_SP2C_2:
                               channel) {
7   symbols
8     AnyC        : agent;
9     Resource    : http_element;
10    Nonce       : int;
11    Any         : hmac;
12  % [...]
13 }
```

The behavior of the entity `ServiceProvider` is shown below. Upon the reception of the client's request containing its private identifier `identifier(Actor)`, the `SP` will discover towards which OpenID provider `IdP` it must request the client to authenticate to by querying the `Discovery` set for the pair `AnyC, IdP`. Next, `SP` redirects `C` to authenticate to `IdP` including in the `httpResp` the `aReq`, embedding SP's handle. After authenticating to `IdP`, the client presents a signed assertion to `SP`. As only the `Shared_key` between `SP` and `IdP` can be used to compute the `hmac(Shared_key, IdP, ?AnyC, Actor, ?Nonce, Handle)` the verification is straightforward. The `Nonce` is then added to the set of consumed values. In the last step, provided that all conditions are satisfied, the resource is delivered to `C`.

Listing 4.6: Interaction sample.

```
1  body {
2    select{
3    on(?C  -Ch_C2SP_1-> Actor: httpReq (post, URI, nil_http_element,
4                                                    identifier(?AnyC)) &
5       Discovery->contains((?AnyC, ?IdP))): {
6        Actor -Ch_SP2C_1->   C: httpResp(code_30x, IdP, aReq(AnyC, IdP,
             Handle,
7                                                       Actor),
                                                            nil_http_element
                                                            );
8        select{
9          on(?C -Ch_C2SP_2-> Actor: httpReq(get, Actor, aResp(IdP, ?AnyC
               , Actor,
10                                       ?Nonce, Handle, hmac(Shared_key,
                                             IdP,
11                                       ?AnyC, Actor, ?Nonce, Handle)),
12                                       nil_http_element) &
13             !ConsumedNonces->contains((?Nonce))): {
14               ConsumedNonces->add(Nonce);
```

```
15              Resource := fresh();
16              Actor -Ch_SP2C_2->   C: httpResp(code_200, nil_agent, LID
                    ,
17                                          nil_http_element, Resource);
18          }
19        }
20      }
21    }
22  }
```

Sessions and Environment

The scenarios that we considered for the analysis involve a trusted identity provider, two service providers, one of which malicious, and a client. In our analysis we suppose two concurrent protocol execution, one with an honest `sp` and another where it is instantiated with `i`. Notice that `i` shares a valid secret with `idp`.

Listing 4.7: Sessions and environment models.

```
1  entity Environment {
2    symbols
3      % Protocol Message
4
5    entity Session ( % Parameters
6                     ) {
7      body {
8        new ServiceProvider (SP, IdP, C, URI,
9                     Discovery, ConsumedNonces, Shared_key, Handle,
10                    Ch_C2SP_1, Ch_SP2C_1,
11                    Ch_C2SP_2, Ch_SP2C_2);
12       new IdentityProvider (IdP, SP, C,
13                    Shared_key, Handle,
14                    Ch_C2IdP, Ch_IdP2C);
15       new Client (C, IdP, SP, URI,
16                    Ch_C2SP_1, Ch_SP2C_1,
17                    Ch_C2IdP, Ch_IdP2C,
18                    Channels);
19     } goals
20         %% G1 and G2 goals
21   }
22
23   body {
24     Discovery      := { (c, idp), (i, idp) };
```

```
25      ConsumedNonces := { };
26      CChannels := {(sp, ch_c2sp_2s1, ch_sp2c_2s1), (i, ch_c2i_2s2,
            ch_i2c_2s2)};
27          %% Two protocol runs
28          % honest agents
29      new Session(c, idp, sp, uri_sp,
30                  Discovery, ConsumedNonces, hmac_key_sp, handle_sp_idp,
31                  ch_c2sp_1s1, ch_sp2c_1s1,
32                  ch_c2sp_2s1, ch_sp2c_2s1,
33                  ch_c2idp_s1, ch_idp2c_s1,
34                  CChannels);
35      % malicious sp (==i)
36      new Session(c, idp, i , uri_i ,
37                  Discovery, ConsumedNonces, hmac_key_i,  handle_i_idp ,
38                  ch_c2i_1s2, ch_i2c_1s2,
39                  ch_c2i_2s2, ch_i2c_2s2,
40                  ch_c2idp_s1, ch_idp2c_s1,
41                  CChannels);
42    }
43  }
```

Security goals

As we discussed in Chapter 3, we expect that OpenID satisfies the the mutual authentication property G1 and the confidentiality of the resource G2. We specified the goal as seen for SAML SSO in 4.1.2.

4.2 Formal Analysis

In this section we present the formal analysis of SAML SSO and OpenID. This section is organized as follows. First, in Section 4.2.1 we give an introduction to the AVANTSSAR platform. Then, in Section 4.2.2 and Section 4.2.3 we discuss the options we considered and show the results.

4.2.1 The AVANTSSAR Platform

The architecture of the AVANTSSAR platform is shown in Figure 4.1. The AVANTSSAR platform takes as input a high-level specification of a security protocol, the expected security goals, as well as the scenario in which

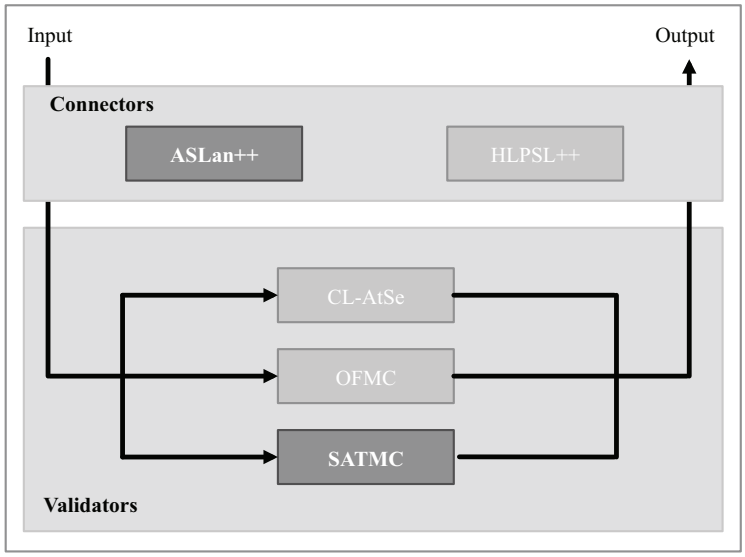


Figure 4.1: The AVANTSSAR Platform

the protocol is employed, and automatically evaluates its security. The AVANTSSAR platform supports two highlevel speficiation languages, namely HLPSL++ and ASLan++. The high-level specification is translated by a connector into an intermediate specification language amenable to formal analysis. The intermediate specifications feed the validator which automatically checks whether the protocol achieves its security goals. If this is not the case, then an attack trace is returned and translated back into a user-friendly format. Currently the AVANTSSAR platform supports three model checker for security protocols, namely CL-AtSe [Tur06b], OFMC [MV09], and SATMC [ACC07].

In the experiments reported in this chapter, we used the ASLan++ connector and the SATMC back-end. The ASLan++ connector takes as input an ASLan++ specification and translates it to ASLan, the intermediate specification language. In addition, the ASLan++ connector displays attacks (if any) as message sequence charts (MSC). SATMC takes as input a formal specification, a scenario to be considered for the analysis, a specification

of the expected security property, and an integer `max`. SATMC determines whether the protocol satisfies the expected security property in the scenario by considering up to `max` execution steps. At the core of SATMC lies a procedure that automatically generates a propositional formula whose satisfying assignments (if any) correspond to attack of length bounded by some integer $k \leq$ `max`. Therefore, finding attacks (of length k) on the protocol boils down to solving propositional satisfiability problems. SATMC relies on SAT solvers for this task which can handle propositional satisfiability problems with hundreds of thousands variables and clauses and even more. SATMC can also be instructed to perform iterative deepening on k. By setting `max` to infinite (`max` $= -1$), SATMC is a semi-decision procedure that it is guarantee to terminate if there is an attack, but may not terminate if the protocol is secure. SATMC is a decision procedure for protocols without loops, i.e. it is guaranteed to terminate with a definitive sound answer. The security protocols considered in this chapter do not have loops and thus fall in this decidable class. When run against them, SATMC is thus guaranteed to either report an attack (if any) or to reach a termination condition that ensures that enough execution steps, say k_{safe}, have been explored proving the safety of the protocol (i.e., absence of attacks).

4.2.2 SAML SSO

Table 4.1 and Table 4.2 show the results of the analysis. Each entry is a model. The column *MID* is the unique identifier of the model. The column *from* is the MID it derives from. The remaining columns are the options organized in four areas. The first area is the use of the ARP. As we discussed, ARP can be used in two distinct phases of SAML SSO for exchanging the authentication request *AReq* and for the authentication response *AResp*. In the IdP-initiated profile, ARP is used only for the latter. The second area is the use of secure transport layer. The SAML specification recommends the use of secure transport layer such as SSL/TLS for carrying authentication requests and authentication assertions. However, when SSL/TLS is not used for transmitting *AResp*, the protocol is trivially vulnerable to the Man-In-

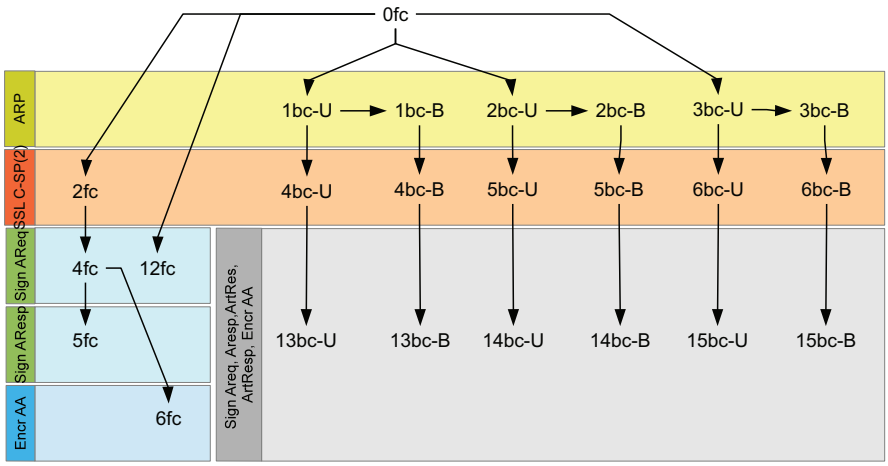


Figure 4.2: Configurations for the SP-initiated profile

The-Middle attack in which the attacker overhear the assertion and reuses it to impersonate C. In our analysis we assumed that SSL/TLS is always used for exchanging *AResp* and consider optional the use of SSL/TLS for exchanging *AReq*. When back channels are used, we assume that SSL/TLS is always used (columns *ARP: AReq-ARP: AResp*). However, we explored two types of SSL/TLS connection; server-side unilateral authenticated SSL/TLS and bilateral authenticated SSL/TLS channels, respectively identified by "U", and "B" in Table 4.1 and Table 4.2. The third area is the use of signature to digitally sign SAML messages. The SAML SSO profiles we considered count in total five SAML messages that can be signed: the authentication request, the authentication response, the assertion, the artifact resolve request, and artifact response. The SAML specification mandates that the assertion is always signed leaving the signature for the other 4 messages optional. The last area is the use of encryption for encrypting portion of messages. In this analysis we considered the encryption of the authentication assertion in the the SP-initiated profile.

The last two columns show whether an attack to the properties has been discovered by the model checker. In Table 4.1 and Table 4.2 we use the

		ARP		SSL/TLS			Sign					Attacks	
MID	from	AReq	AResp	C-SP:AReq	ARP: AReq	ARP:AResp	AReq	AResp	ArtResolve	ArtResp	Encr. AA	G1	G2
0fc	-	n	n	n	-	-	n	n	-	-	n	y	n
1bc-U	0fc	y	n	n	U	-	n	n	n	-	n	y	n
2bc-U	0fc	n	y	n	-	U	n	n	-	n	n	y	n
3bc-U	0fc	y	y	n	U	U	n	n	n	n	n	y	n
1bc-B	1bc-U	y	n	n	B	-	n	n	n	-	n	y	n
2bc-B	2bc-U	n	y	n	-	B	n	n	-	n	n	y	n
3bc-B	3bc-U	y	y	n	B	B	n	n	n	n	n	y	n
4bc-U	1bc-U	y	n	y	U	-	n	n	n	-	n	y	n
5bc-U	2bc-U	n	y	y	-	U	n	n	-	n	n	y	n
6bc-U	3bc-U	y	y	y	U	U	n	n	n	n	n	y	n
4bc-B	1bc-B	y	n	y	B	-	n	n	n	-	n	y	n
5bc-B	2bc-B	n	y	y	-	B	n	n	-	n	n	y	n
6bc-B	3bc-B	y	y	y	B	B	n	n	n	n	n	y	n
13bc-U	4bc-U	y	n	y	U	-	y	y	y	y	y	y	n
14bc-U	5bc-U	n	y	y	-	U	y	y	y	y	y	y	n
15bc-U	6bc-U	y	y	y	U	U	y	y	y	y	y	y	n
13bc-B	4bc-B	y	n	y	B	-	y	y	y	y	y	y	n
14bc-B	5bc-B	n	y	y	-	B	y	y	y	y	y	y	n
15bc-B	6bc-B	y	y	y	B	B	y	y	y	y	y	y	n
2fc	0fc	n	n	y	-	-	n	n	-	-	n	y	n
4fc	2fc	n	n	y	-	-	y	n	-	-	n	y	n
5fc	4fc	n	n	y	-	-	y	y	-	-	n	y	n
6fc	4fc	n	n	y	-	-	y	n	-	-	y	y	n
12fc	0fc	n	n	n	-	-	y	n	-	-	n	y	n

Table 4.1: Results for the SP-initiated profile

symbol “y” when the option is used, “n” when the option is not used, the symbol “-” for option non applicable, “U” for unilateral server authenticated SSL/TLS channel, and “B” for bilateral authenticated SSL/TLS channel.

We derived the formal specification as shown in Figure 4.2. We started from a single, initial configuration for each profile, i.e. 0fc, and we derived new models by enabling one option per time. The initial configuration is

		ARP	SSL/TLS	Sign		Attacks	
MID	from	AResp	ARP: AResp	AA	AResp	G1	G2
0fc	-	n	-	y	n	n	n
1bc-U	0fc	y	U	y	n	n	n
1bc-B	1bc-U	y	B	y	n	n	n
2bc	1bc-U	y	n	y	n	n	n

Table 4.2: Results for the IdP-initiated profile

taken from the prototypical examples available in the SAML specifications. In total, we wrote 28 formal models.

In our experiments, we used the ASLan++ connector and the SATMC validator. Table 4.1 and Table 4.2 shows the following results. First, the property G1 is achieved by all the model both SP- and IdP-initiated. Second, the IdP-initiated models achieve the authentication property G1. Third, the SP-initiated SAML SSO does not satisfy the property G1. Fourth, by enabling the protocol options G1 is never satisfied.

4.2.3 OpenID

Table 4.3 shows the results of the analysis of OpenID. Each entry is a model. The column *MID* is the unique identifier of the model, and the column *from* points to the model it derives from. The remaining columns are the options and the result of the formal analysis. The column *C-SP: AReq* is the use of SSL/TLS communication channels in steps A1 and A2, whereas the column *C-SP: AResp* refers to the use of SSL/TLS communication channels for the steps A4 and A5. The last two columns are for the results of the model checker for the property G1 and G2.

As seen for SAML SSO, we derived the specifications starting from an initial configuration and then we added incrementally the other options. We wrote in total 4 models of OpenID. The setup for our tests is the same used for SAML SSO.

		SSL/TLS		Attacks	
MID	from	C-SP: AReq	C-SP: AResp	G1	G2
0	-	n	n	y	y
1	0	y	n	y	y
2	1	y	y	y	n

Table 4.3: Results for OpenID

Table 4.3 shows the following results. First, when SSL/TLS is not used for delivering the AResp to the SP, then the property G2 is not achieved. The attack violating the property G2 is trivial: the attacker (i) overhears the AResp that C sends to SP, (ii) blocks the delivery of the message to SP, and (iii) sends the AResp at the SP. The second result is that G1 is never satisfied.

4.3 Logic Flaws

4.3.1 SAML SSO

Table 4.1 shows that the SAML SSO SP-initiated does not satisfy the property G1. A closer look at the the counterexamples revealed that them expose a common attack pattern. The attack pattern is in Figure 4.3.

The attack involves four principals: a client (c), an honest IdP (idp), an honest SP (sp) and a malicious SP (i). The attack is carried out as follows: c initiates the protocol by requesting a resource uri_i at SP i. Now i, pretending to be c, requests a different resource uri at sp and sp reacts according to the standard by generating an Authentication Request, which is then returned to i. Now i maliciously replies to c by sending an HTTP redirect response to `idp` containing $\mathit{AReq}(\text{id}, \text{sp})$ and uri (instead of $\mathit{AReq}(\text{id}_\text{i}, \text{i})$, 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

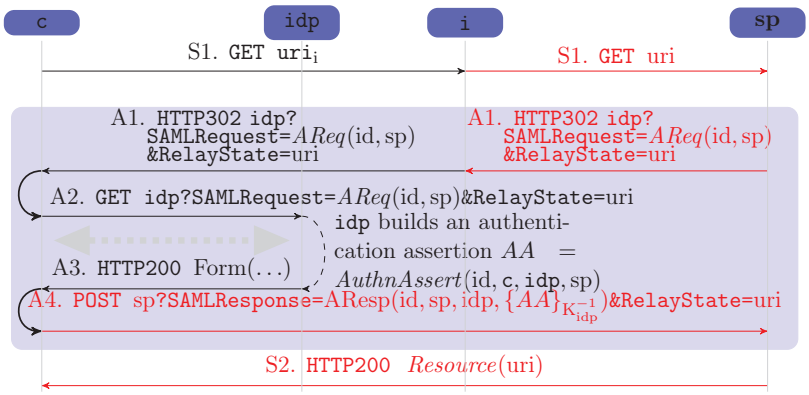


Figure 4.3: Authentication Flaw of the SAML 2.0 Web Browser SSO Profile

asked for a resource from i.

The attack in Figure 4.3 does not strictly require a malicious SP in order to be successful. Any malicious web server i would be able, upon a request from c, to mount the attack provided that *(i)* c is a client of sp and *(ii)* c has an active authentication context with idp. The attack in Figure 4.3 can be exploited in a number of ways:

Delivery of an unrequested resource. The most trivial exploitation of the flaw consists in the attacker forcing the client to receive a protected resource different from the one that was initially requested. The same exploitation may also be mounted if a malicious web server redirects the browser to a legitimate SP before SAML SSO starts. However this attack can be prevented by using well-known browser-side plugins that restrict HTTP redirections (e.g., the NoRedirect addon for Firefox). By allowing only IdP-to-SP and SP-to-IdP redirections, the delivery of an unrequested resource upon redirection outside of the SAML SSO Protocol is prevented, but a malicious SP can still mount the one depicted in the Figure 4.3.

Launching pad for Cross-Site Request Forgery (CSRF) attacks. This attack assumes that the URI that was initially requested did not point to a resource, but rather contained a URL-encoded command, such as a

request for the change of some settings or user's preferences, for the deletion of some resource, or for the annulment of/committing to an action, such as the purchase of a paid good. Depending on the output provided by the execution of the command, the client may or may not be able to detect the attack. This type of attack is even more pernicious than classic CSRF, because CSRF requires C to have an active session with SP, whereas in this case, the session is created automatically hijacking C's authentication attempt.

Launching pad for Cross-Site Scripting (XSS) attacks. It is straightforward to see that this attack also constitutes a launching pad to reflected XSS attacks, i.e. XSS attacks that can be triggered by visiting a maliciously-crafted URL. In addition, a vanilla implementation of the SAML SSO protocol exposes the `RelayState` field to a possible injection of malicious code that may be executed at the honest SP side. Although the SAML standard recommends to protect the integrity of this field, our experience shows that this often is not the case (see Section 4.4.1).

4.3.2 OpenID

Table 4.3 shows that the property G1 is not satisfied. The counterexamples returned by the model checker have a common attack pattern. The attack pattern is shown in Figure 4.4

The attack involves four principals: a client (c), an honest IdP (idp), an honest SP (sp) and a malicious service provider (i). The client c requests uri_i at SP i. Here, the attacker i, impersonating c, requests a different resource to sp and sp reacts starting OpenID by crafting a proper authentication request for c. The malicious SP i uses this authentication request in its protocol session with c. The protocol simply proceeds according to the OpenID standard resulting in c accessing to a resource of sp, while c originally asked for a resource from i.

The differences between SAML SSO and OpenID make the exploitations on SAML SSO not directly applicable to OpenID. For example, OpenID does

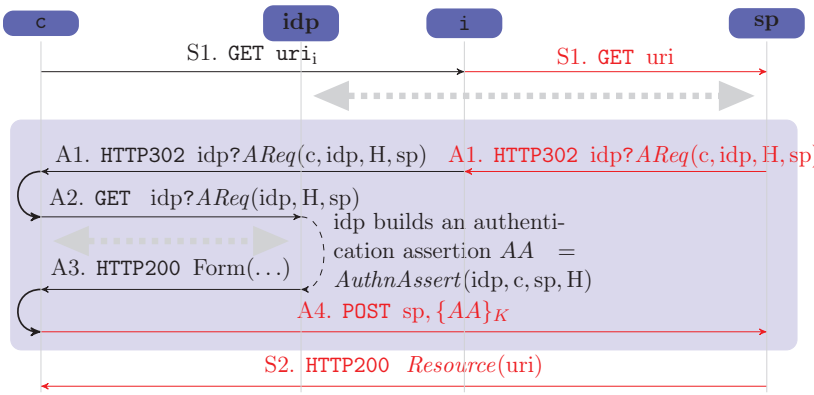


Figure 4.4: Authentication Flaw of the OpenID SSO Protocol

not prescribe any parameters to let SP recover its previous state (e.g. `RelayState` in SAML SSO). Therefore, the CSRF and the XSS attacks described in Section 4.3.1 are not possible.

4.4 Testing Real Implementations

In the previous sections, we described the application of a model checking technique to the security analysis of authentication protocols. Our analysis led to the discovery of an attack to the mutual authentication property G1. Moreover, we discussed possible exploitations of the flaw. However, the AVANTSSAR tool and, in general, model checking techniques, offer no support for testing real implementations. As a consequence, we tested manually protocol implementations for verifying whether the attack returned by the model checker is applicable. Then, we verified possible exploitations of the flaw.

This section reports on the results of manual testing SAML SSO and OpenID implementations.

4.4.1 Exploitations in SAML SSO

We have analyzed a number of SAML-based SSO solutions available on the market, including the SAML-based SSO for Google Apps, a deployment of Novell Access Manager v.3.1, and SimpleSAMLphp as deployed for Foodle (`https://foodl.org`). All these deployments support the SAML SSO use case. As expected, by inspecting the messages exchanged between the parties we verified that SPs accept and process SAML responses carried over SSL/TLS channels different from that used to deliver the SAML request.

The SAML-based SSO for Google Apps Our analysis of the SAML-based SSO for Google Apps shows that by exploiting the weakness we discovered with the model checker, a compromised SP can force C to consume a resource from Google, e.g., by visiting any page of the GMail service. This trivial attack is however easily detected by the user using C, and does not bring any real advantage to the attacker. Definitely more serious was the XSS attack we were able to execute and that allowed the compromised SP to steal the cookies of C for the Google domain and thus to impersonate C on any Google application. The abstract flaw of Figure 4.3 served indeed as launching pad for this XSS. The attack is depicted in Figure 4.5. As we can see in the figure, c requires a resource from a compromised SP i; i, acting in turn as a client, receives from sp an Authentication Request, and passes it back to c, with the malicious code injected into the `RelayState`. The client's browser eventually executes the redirection to the maliciously-crafted URI, as if coming from the Google domain (thus circumventing the same origin policy). This redirection leads to the theft of the session cookies by sp. In other words, the combination of the abstract flaw and the missing sanitization was key to this XSS attack. In response to our vulnerability report Google patched the issue by properly sanitizing the `RelayState` value. An acknowledgement of our contribution can be found in the Google corporate web pages [Goo09].

Legenda: ·····➤ : `https`

Figure 4.5: XSS Attack on the SAML-based SSO for Google Apps

Novell Access Manager We have also analyzed the SAML SSO solution of the Novell Access Manager v.3.1 as deployed in a real industrial environment and even in this case we were able to confirm the authentication flaw. We have been able to mount a XSS attack similar to the one found in the Google SSO solution. In this deployment `RelayState` is not used to store the URI; instead, a URL-encoded parameter is used to this end and also in this case, the parameter was not sanitized. In response to our findings Novell promptly patched their implementation and issued a vulnerability report [Nov11].

SimpleSAMLphp SimpleSAMLphp, as deployed in Foodle, is not vulnerable to the authentication flaw that we discovered with the model checker. The reason is that SPs running SimpleSAMLphp additionally use cookies that block the flaw. SimpleSAMLphp stores the initially requested URI into the URL parameter `ReturnTo`. Although that field is not sanitized, we have not been able to mount any XSS.

Also in this case we promptly informed the developer and maintainer of the SSO solution, namely UNINETT. UNINETT credited us in the release notes of a new version of SimpleSAMLphp [UNI10].

4.4.2 Exploitation in OpenID

The authentication flaw on SAML SSO can be exploited in several ways. However, the differences between SAML SSO and OpenID make the exploitations on SAML SSO not directly applicable to OpenID. For example, OpenID does not prescribe any parameters to let SP recover its previous state (e.g. `RelayState` in SAML SSO). Therefore, the CSRF and the XSS attacks described in Section 4.3.1 are not possible. However, the OpenID specifications enable SP to append customized parameters to the redirection URLs whose names and values are out of the scope of the OpenID specifications. Depending on their use, they can be exploited in a similar way as we have seen for SAML SSO.

We have verified that the Zoho Invoice service provider [3] used with the OpenID provider by Google or by Yahoo suffers from the logic flaw of Section 4.3.2.

4.5 Conclusions

In this chapter, we presented the formal analysis of SAML SSO and OpenID via model checking. Starting from the protocol specifications, we formalized the seven protocol flows as well as the security-relevant configuration options. We verified the different configurations discovering a previously

[3] `http://invoice.zoho.com`

unknown logic flaw allowing an attacker to mount CSRF attacks or cause the delivery of unrequested resources. We tested manually five real protocol implementation against the counterexample returned by the model checker. Four out of five implementations suffer from the authentication flaw. Moreover, we discovered that in presence of XSS vulnerabilities, an attacker can use the logic flaw as a lunching pad for XSS attacks in which the attacker hijacks the user session by stealing the session cookies.

In this chapter we showed that model checking is a powerful technique for detecting logic flaws in security protocols specifications. However, the counterexamples prove that the model does not satisfy a given security protocol, while nothing is said about the implementations of the protocol. To this end, the counterexample must be interpreted and executed against the implementations deployed on the wild. However, reproducing counterexamples not only requires a thorough understanding of both the protocol and its implementation, but also a substantial amount of manual activity. In Chapter 5 we present a technique that tackles these difficulties.

Chapter 5
From Model Checking to Security Testing

In Chapter 4, we showed that when specifications are available, model checking can be used to detect subtle flaws in the logic of the application. However, the counterexamples returned by the model checker witness a violation of a security property in the model, and, which does not necessarily reflect a vulnerability in a real implementation. Moreover, the model checker provides little support for testing the real implementations. As a result, counterexamples are normally interpreted and executed manually against the real system. In this chapter we propose an automatic model checking-driven approach for testing security protocols against counterexamples returned by the model checker. We applied our technique to four SAML SSO and OpenID protocol implementations. The experiments show that the approach is capable of detecting the logic flaws of SAML SSO and OpenID into real implementations.

Structure: This chapter is organized as follows. Section 5.1 describes the architecture of the approach. Section 5.2 presents the ASLan language. Section 5.3 describes the instrumentation techniques. Then, in Section 5.4 we describe the test execution engine. Section 5.5 shows the experiments and the results of our tests. Finally, Section 5.6 draws some conclusions.

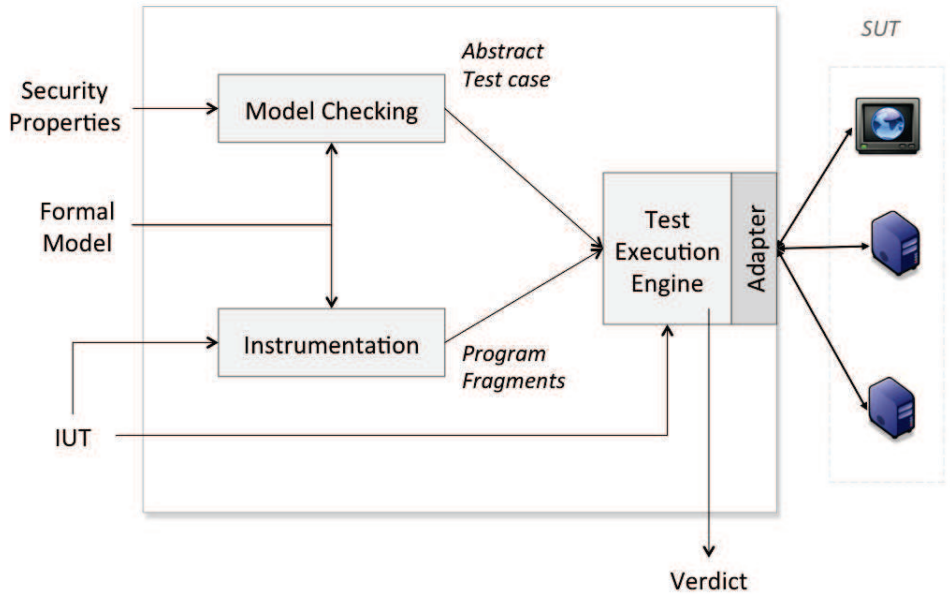


Figure 5.1: Overview of the Approach

5.1 Architecture

An overview of our approach is shown in Figure 5.1. It takes in input a model amenable for formal analysis, a security property, and the implementation under test IUT. The IUT is a data structure containing the *mapping* between abstract model symbols and real values and the protocol participants that are under test. Our approach consists of the following steps:

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 *abstract test case*.

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* in the IUT input.

Execution The *Test Execution Engine* (TEE) interprets the *abstract test case* and executes the program fragments accordingly. The IUT 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).

5.2 Model Checking

In this thesis, we used ASLan++ to model SAML SSO and OpenID. As said in Section 4.2.1, the ASLan++ connector translates ASLan++ specifications into ASLan, an intermediate language amenable for formal analysis. In this section we present a simplified version of ASLan, featuring only the aspects of the language that are relevant for this work.

Background of the AVANTSSAR Platform

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 [ACC+08]. The states of M are sets of ground (i.e. variable-free) *facts*, i.e. atomic formulae of the form given in Table 5.1.

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

[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.

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.

Table 5.1: Facts and their informal meaning

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 5.2.1 and 5.2.2. 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 lowercase letter (e.g. `client`, `hReq`).

Message modeling Messages are represented as follows. HTTP requests are represented by expressions $\texttt{httpReq}(mtd, addr, qs, body)$, 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 $\texttt{hResp}(code, loc, qs, body)$, 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 3.1 is

$$\texttt{hResp}(\texttt{c30x}, \texttt{IdP}, \texttt{hBind}(\texttt{aReq}(\texttt{SP}, \texttt{IdP}, \texttt{id}(\texttt{N})), \texttt{URI}), \texttt{nil})$$

obtained by composing `hResp`, `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.

The above expressions are sufficient to capture the relevant aspects of SAML SSO. However, it must be noted that it can be adjusted by adding or removing parameters according to the need. For example, `httpReq` and `hResp` can be extended with further parameters in order to support basic authentication HTTP headers [FHBH+99] as used by OAuth 2.0 [Har12].

5.2.1 Specification of the rules of the honest agents

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

$$\mathtt{sent}(b_{rs}, b_i, a, m_i, c_i) \centerdot \mathtt{state}_r(j, a, [e_1, \ldots, e_p]) \xrightarrow{\mathtt{send}_r^{j,k}(a,\ldots)} \mathtt{sent}(a, a, b_o, m_o, c_o) \centerdot \mathtt{state}_r(l, a, [e'_1, \ldots, e'_q]) \quad (5.1)$$

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 (5.1) 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 $\mathtt{sent}(\ldots)$ is omitted in the left and right hand sides of the rule (5.1), respectively. For instance, the rule for receiving the message A1 and sending message A2 in Figure 3.1 is modeled as follows:

$$\begin{aligned}
&\mathtt{sent}\,(\mathtt{SP1}, \mathtt{SP}, \mathtt{C}, \mathtt{hResp}(\mathtt{c30x}, \mathtt{IdP}, \mathtt{AReq}, \mathtt{nil}), \mathtt{C}_{\mathtt{SP2C}}) \centerdot \\
&\quad \mathtt{state}_\mathtt{c}(2, \mathtt{C}, [\mathtt{SP}, \mathtt{IdP}, \mathtt{URI}, \mathtt{C}_{\mathtt{C2SP}}, \mathtt{C}_{\mathtt{SP2C}}, \mathtt{C}_{\mathtt{C2SP}_2}, \mathtt{C}_{\mathtt{SP2C}_2}, \mathtt{C}_{\mathtt{C2IdP}}, \mathtt{C}_{\mathtt{IdP2C}}]) \\
&\qquad \xrightarrow{\mathtt{send}_\mathtt{c}^{2,1}(\mathtt{C},\mathtt{IdP},\mathtt{SP},\mathtt{SP1},\mathtt{URI},\mathtt{AReq},\mathtt{C}_{\mathtt{C2SP}},\mathtt{C}_{\mathtt{SP2C}},\mathtt{C}_{\mathtt{C2SP}_2},\mathtt{C}_{\mathtt{SP2C}_2},\mathtt{C}_{\mathtt{C2IdP}},\mathtt{C}_{\mathtt{IdP2C}})} \\
&\mathtt{state}_\mathtt{c}(4, \mathtt{C}, [\mathtt{SP}, \mathtt{IdP}, \mathtt{URI}, \mathtt{AReq}, \mathtt{C}_{\mathtt{C2SP}}, \mathtt{C}_{\mathtt{SP2C}}, \mathtt{C}_{\mathtt{C2SP}_2}, \mathtt{C}_{\mathtt{SP2C}_2}, \mathtt{C}_{\mathtt{C2IdP}}, \mathtt{C}_{\mathtt{IdP2C}}]) \centerdot \\
&\qquad \mathtt{sent}\,(\mathtt{C}, \mathtt{C}, \mathtt{IdP}, \mathtt{hReq}(\mathtt{get}, \mathtt{IdP}, \mathtt{AReq}, \mathtt{nil}), \mathtt{C}_{\mathtt{C2IdP}}) \quad (5.2)
\end{aligned}$$

5.2.2 Specification of the rules of the intruder

The abilities of the DY intruder to intercept and overhear messages are modeled by the following rules:

$$\begin{array}{lll} \mathtt{sent}(\mathtt{A},\mathtt{A},\mathtt{B},\mathtt{M},\mathtt{C}) & \xrightarrow{\mathtt{intercept}(\mathtt{A},\mathtt{B},\mathtt{M},\mathtt{C})} & \mathtt{ik}(\mathtt{M}) \\ \mathtt{sent}(\mathtt{A},\mathtt{A},\mathtt{B},\mathtt{M},\mathtt{C}) & \xrightarrow{\mathtt{overhear}(\mathtt{A},\mathtt{B},\mathtt{M},\mathtt{C})} & \mathtt{ik}(\mathtt{M}) \,.\, LHS \end{array} \tag{5.3}$$

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 [MCJ97]. The decomposition capabilities of the intruder are modeled by the following rules:

$$\mathtt{ik}(\{\mathtt{M}\}_k) \,.\, \mathtt{ik}(k^{-1}) \xrightarrow{\mathtt{decrypt}(\mathtt{M},\ldots)} \mathtt{ik}(\mathtt{M}) \,.\, LHS \tag{5.4}$$

$$\mathtt{ik}(\{\mathtt{M}\}^s_{\mathtt{K}}) \,.\, \mathtt{ik}(\mathtt{K}) \xrightarrow{\mathtt{sdecrypt}(\mathtt{K},\mathtt{M})} \mathtt{ik}(\mathtt{M}) \,.\, LHS \tag{5.5}$$

$$\mathtt{ik}(f(\mathtt{M}_1,\ldots,\mathtt{M}_n)) \xrightarrow{\mathtt{decompose}_f(\mathtt{M}_1,\ldots,\mathtt{M}_n)} \mathtt{ik}(\mathtt{M}_1) \,.\, \ldots \,.\, \mathtt{ik}(\mathtt{M}_n) \,.\, LHS \tag{5.6}$$

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 the composition rules we consider an optimisation [JRV00] 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 [AC02]. For each protocol rule (5.1) in Section 5.2.1 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:

$$\texttt{ik}(m_{1,l})\texttt{.}\ldots\texttt{.ik}(m_{ji,l})\texttt{.state}_r(j,a,[e_1,\ldots,e_p]) \xrightarrow{\texttt{impersonate}_r^{j,k,l}(\ldots)} \texttt{sent}(\texttt{i},b_i,a,m',c_i)\texttt{.ik}(m')\texttt{.}\mathit{LHS} \quad (5.7)$$

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'.

5.2.3 Specification of the authentication property

The security goal of 4.1.2 are translated by the ASLan++ translator into an linear temporal logic formula. ASLan uses facts propositions as atoms of the formulas, logic operators such as $\wedge$, $\Rightarrow$), the first-order quantifiers $\forall$ and $\exists$, and the temporal operators **F** (eventually), **G** (globally), and **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 [ACC+08] 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* [Low97]. 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:

$$\mathbf{G}\,\forall(\texttt{state}_{\texttt{sp}}(7,\texttt{SP},[\texttt{C},\ldots,\texttt{URI},\ldots]) \Rightarrow \exists\,\mathbf{O}\,\texttt{state}_{\texttt{c}}(2,\texttt{C},[\texttt{SP},\ldots,\texttt{URI},\ldots])) \quad (5.8)$$

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 $\mathtt{c}$ 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 $\mathtt{a}$, 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 $\mathtt{sp}$ is not observable, while the channel $\mathtt{c_{SP2C}}$ is observable, we rewrite the property (5.8) as follows:

$$\mathbf{G}\,\forall(\mathtt{sent}(\mathtt{SP}, \mathtt{SP}, \mathtt{C}, \mathtt{res}(\mathtt{URI}), \mathtt{c_{SP2C}}) \Rightarrow \exists\,\mathbf{O}\,\mathtt{state_c}(2, \mathtt{C}, [\mathtt{SP}, \ldots, \mathtt{URI}, \ldots])) \quad (5.9)$$

where $\mathtt{res}(\mathtt{URI})$ represents the resource returned by $\mathtt{SP}$ in step 7.

5.3 Instrumentation

The model instrumentation aims at calculating program fragments p associated to each rule of the model. Program fragments are then evaluated and executed by the TEE (See Section 5.4) 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 5.2, messages in the formal model are specified abstractly. For instance, let us consider the following SAML authentication request:

```
<AuthnRequest ID="IDreq" Version="2.0" IssueInstant="IIreq"
```

```
  Destination="DS" AssertionConsumerServiceURL="ACS"
  ProtocolBinding="HTTP-POST">
  <Issuer>IS</Issuer>
</AuthnRequest>
```

where 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 URL) is the end-point of the SP. The above SAML request 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 SP. Let D be the set of data values the messages exchanged and their fields. For instance, if $AReq(is, ds, ii, acs, id)$ 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 $\alpha(\overline{f}(d_1, \ldots, d_n)) = f(\alpha(d_1), \ldots, \alpha(d_n))$ 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 as above, replacing $inv(k)$ 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 5.1). 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 computed 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 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. $d = \text{eval}(p)$. 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.

5.3.1 Instrumentation of the rules of the honest agents

Example 5.3.1. Let us consider the following example of ASLan rule:

$$\begin{aligned}&\mathtt{sent}\,(\mathtt{A},\mathtt{A},\mathtt{B},\mathtt{f}(\{\mathtt{g}(\mathtt{A},\mathtt{B},\mathtt{m})\}^{\mathtt{s}}_{\mathtt{K}},\{\mathtt{h}(\mathtt{A},\mathtt{K})\}_{\mathtt{Kb}}),\mathtt{C_{A2B}})\,.\\ &\qquad \mathtt{state_b}(1,\mathtt{B},[\mathtt{B},\mathtt{Kb},\mathtt{inv}(\mathtt{Kb}),\mathtt{m},\mathtt{C_{A2B}},\mathtt{C_{B2A}}]) \xrightarrow{\mathtt{send}^{1,1}_{\mathtt{b}}(\mathtt{B},\mathtt{A},\mathtt{Kb},\mathtt{K},\mathtt{C_{A2B}},\mathtt{C_{B2A}})}\\ &\qquad\qquad \mathtt{state_b}(2,\mathtt{B},[\ldots,\mathtt{A},\mathtt{K}])\,.\,\mathtt{sent}\,(\mathtt{B},\mathtt{B},\mathtt{A},\mathtt{f}(\mathtt{B},\mathtt{m}),\mathtt{C_{B2A}})\end{aligned} \quad (5.10)$$

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 (5.1) 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 $m_o = \alpha(\ell_{m_o})$. In the sequel, we describe how to generate automatically p_{m_i} and p_{m_o} for a generic ASLan rule (5.1).

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 (5.1), 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 $j = 1, \ldots, n$.

Example 5.3.2. Let us provide an example of closure. With reference to the rule (5.10), the set Ms contains the associated expression for the incoming message $\ell_{\mathtt{f(senc(...),enc(...))}} : \mathtt{f}(\mathtt{senc}(K, \mathtt{g}(A, B, \mathtt{m})), \mathtt{enc}(Kb, \mathtt{h}(A, K)))$ 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 $\ell_{\mathtt{f(senc(...),enc(...))}} : \mathtt{f}(\mathtt{senc}(\ldots), \mathtt{enc}(Kb, \mathtt{h}(A, K))) \in Ms \subseteq \downarrow Ms$ then $\pi^1_{\mathtt{f}}(\ell_{\mathtt{f(senc(...),enc}(Kb,\mathtt{h}(A,K)))}) : \mathtt{senc}(\ldots)$ and $\pi^2_{\mathtt{f}}(\ell_{\mathtt{f(senc(...),enc}(Kb,\mathtt{h}(A,K)))}) : \mathtt{enc}(Kb, \mathtt{h}(A, K))$ are in $\downarrow Ms$ (case 4 of the definition). Given that $\ell_{Kb} : Kb$ is in $\downarrow Ms$, the case 2 is applicable, thus $\mathit{decrypt}(\ell_{\mathtt{inv}(Kb)}, \pi^2_{\mathtt{f}}(\ldots)) : \mathtt{h}(A, K) \in \downarrow Ms$ as well.

Example 5.3.2 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 section 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 (5.1), 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` (`!`=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 (5.10), 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` (`!`=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}.

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 (5.1), the program fragment p_{m_o} is calculated by $\text{MsgGen}(m_o)$ over $kn = \{e'_1, \ldots, e'_q\}$.

5.3.2 Instrumentation of the rules of the intruder

Intercept and overhear rules

Let us consider the intercept rule (5.4) in Section 5.2. 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 (5.4) in Section 5.2 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 (5.4) is as follows:

ℓ_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 (5.5). The fragment $p_{\texttt{decompose}_f(M_1,\ldots,M_n)}$ encoding the rule (5.6) is as follows:

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

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.

Composition rules

Let us consider the impersonate rule (5.7) in Section 5.2. The fragment of pseudocode $p_{\texttt{impersonate}_r^{j,k,l}(\ldots)}$ encoding this rule is computed by MsgGen(m') over the knowledge $kn = \{m_{1,l}, \ldots, m_{j_l,l}\}$.

5.4 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, ..., 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       while eval(p_{step_i}) == false do:
5         if handle_error() == false then:
6           printf("Test failed in step %s", step_i);
7           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 the TEE executes an error handling procedure. If the error is correctly handled,

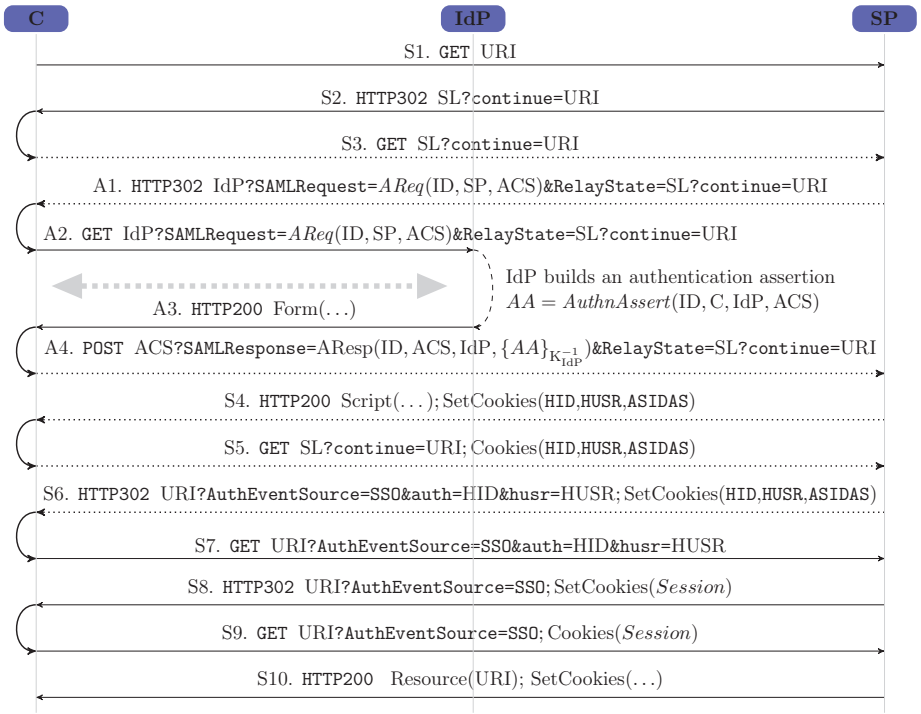
then the test continues, otherwise, (lines (6)–(7)) notifies that an error occurred.

5.4.1 Error Handling

Protocol implementations may differ from the model. For example, SP may perform addition HTTP 30x redirections between the message S1 and A1. In general, when a mismatch between the message received and the message expected is detected, eval(p_{step_i}) returns false. This error is captured by the TEE that in turn executes an error handling procedure. If the error can be handled, e.g., by executing the HTTP redirection, the TEE repeats the step that cause the error until it succeed. If the error cannot be handled, then the TEE interrupts the execution and reports that the test execution failed.

5.5 Experimental Results

In order to assess the effectiveness of the proposed approach, we have developed a tool of the architecture depicted in Figure 5.1. A complete description of the tool is given in Chapter 7. In this section we introduce only the main aspects. We implemented the instrumentation, the TEE and the adapter modules in Java. The model checking module is the SATMC model checker tool [ACC07]. The instrumentation module takes in input an ASLan model and the mapping, and it returns a Java class where each method is a program fragment. The TEE instantiates the class and executes the attack trace as described in Section 5.4. The adapter implements the constructor and selector functions defined in Section 5.3. 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/`). These functions are used by program fragments as described in Section 5.3. We used the attack traces shown in Figure 4.3 and Figure 4.4.



Legenda: ·····➤ : `https`

Figure 5.2: SAML-based Single Sign-On for Google Apps

5.5.1 Protocol Implementations Under Test

In this section we describe two implementations of the profile SP-initiated of SAML SSO, and one implementation of the OpenID protocol used with different IdPs. They are the SAML-based SSO for Google Apps, SimpleSAMLphp, and Zoho Invoice used with Google and Yahoo OpenID identity provider.

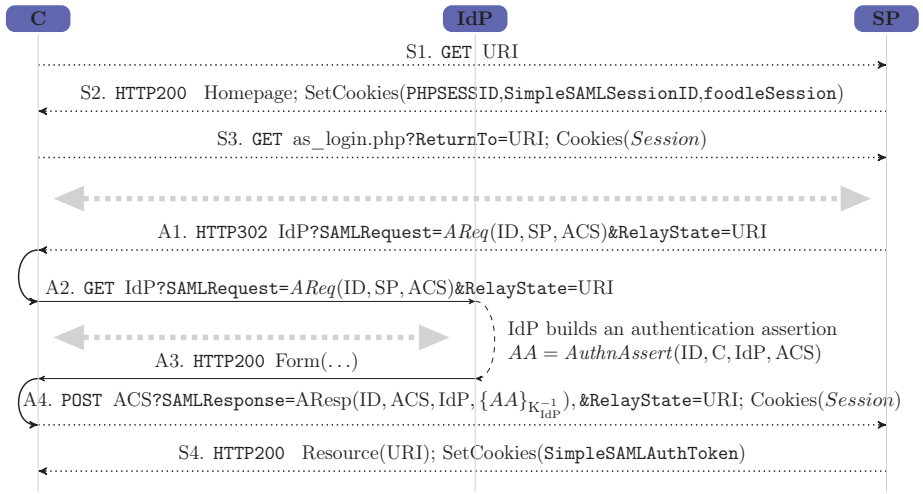
SAML-based SSO for Google Apps

The protocol implemented in the SAML-based SSO for Google Apps in operation until 2009 is depicted in Figure 5.2. The model has been obtained by carefully inspecting the reference implementation of SAML-based Single Sign-On for Google Apps and by experimenting with the online service.

In the implementation offered by Google, when SP receives a request for a resource URI from C, if the request is accompanied by a valid session cookie, then the resource is returned right away (step S10 in Figure 5.2). The name of the session cookie depends on the specific service considered, e.g. it is named `CALH` in case of Google Calendar, and `GXAS` in case of Gmail. If the request is accompanied by valid values for the parameters `auth` and `husr` in the URI, then SP creates a fresh session cookie and sends it back to C; C then is asked to resubmit the request by means of an HTTP redirect (steps S8 and S9). If neither of the above conditions hold, C is redirected to the Service Login (SL) and the requested URI is passed as the value of the `continue` URL-encoded parameter. Upon receipt of this request, SL initiates the SAML Authentication Protocol (step A1) using SL`?continue=`URI as the value of the aforementioned `RelayState` field. If the SAML Authentication Protocol completes successfully, then SL sets the cookies `HID`, `HUSR`, and `ASIDAS` and returns an HTML page of the form (concisely indicated as Script(...) in step S4 of Figure 5.2):

```
<html>
 ...
 <body>
  <script>
   var url=URI
   ...
   window.setTimeout(
         function() {
            window.location = url;
         },
         0);
  </script>
  ...
 </body>
</html>
```

This simulates a redirection by setting the value of the browser variable `window.location` to URI and forcing the browser to reload the page. Notice that since the value of URI is embedded into the HTML page, it will be



Legenda: .....➤ : `https`

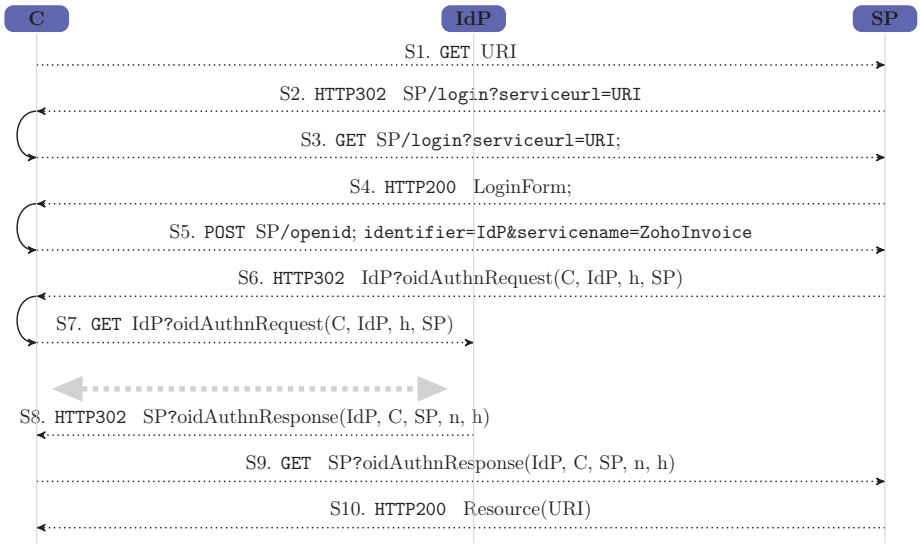
Figure 5.3: SimpleSAMLphp as deployed in Foodle

evaluated by the JavaScript interpreter.

SimpleSAMLphp

The protocol implemented by SimpleSAMLphp is depicted in Figure 5.3 where SP is the Foodle service offered by Uninett and available at `http://foodl.org`, while IdP is the OpenIdP provided by Uninett and available at `https://openidp.feide.no/`.

The protocol execution starts with C asking for the resource URI to SP. In step 2, SP redirects C to an internal login service of SP. Here, between step S3 and A1 takes place the identity provider discovery protocol. The identity provider discovery protocol aims at identitifying the last IdP used by C by inspecting the cookies of C. In all our experiments, we assume that when C starts the protocol, C does not have any cookie installed. As a result the identity provider discovery protocol fails and SP shows to C a list of IdP. C selects IdP, and the SAML protocol begins. In step A1, SP redirects C to the IdP with an authentication request. The IdP challenges C and redirects C to the SP. The protocol ends with SP providing the resource to C.



Legenda: .....➤ : `https`

Figure 5.4: Zoho Invoice relaying party service

Zoho Invoice Relying Party

Our experiments focused on the relying party provided by Zoho Invoice. Zoho Invoice is an online billing solution for small business `https://www.zoho.com/invoice` developed by Zoho. Zoho Invoice service allows users to be registered and log in at their domain. Alternatively, users can login by using SSO protocols such as OpenID, or other protocols such as OAuth2.0. Zoho Invoice supports only two OpenID identity providers, they are Google OpenID and Yahoo OpenID. In our experiments we have focussed only on Zoho Invoice service when used with both OpenID identity providers.

Figure 5.4 shows the HTTP messages exchanged between a web browser guided by an user C, the identity provider IdP, and the Zoho Invoice service SP for accessing the front page URI=`https://invoice.zoho.com/view/ZB_Main/ZB_Invoice`.

In step S1, C requests the resource URI. Then, SP redirects C to the local login service SP/login with a parameter `serviceurl` carrying the original resource asked by C. In step S4, the login service of SP returns the login page

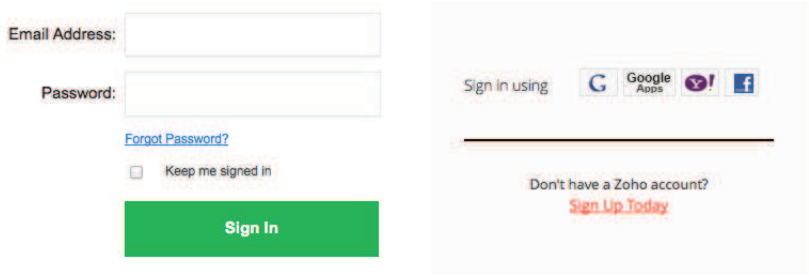


Figure 5.5: User login at Zoho Invoice

of Figure 5.5. Afterwards, C selects to login with Google or Yahoo OpenID identity provider. As a result, SP prepares the authentication requests and redirects C to the IdP selected for being authenticated. The user is then authenticated and, in turn IdP prepares an authentication response. In step S8, IdP redirects C to SP together with the authentication response. Once SP receives the authentication response, SP provides the resource to C.

5.5.2 Experiments

General setup

The first step of the experiments involves the setup the testing environment, in which we registered a user account at each of the identity provider that we tested.

SAML SSO We registered the domain *ai-lab.it* at Google Apps[2]. Then, we deployed our own SAML identity provider in our servers[3]. In the configuration panel of Google Apps, we added our identity provider in the list of trusted identity providers. Moreover, we uploaded the public key for verifying the signature of the assertions. Finally, we created two user accounts for testing; the first one at our identity provider, the second at the OpenIdP by Uninett.

[2]When we set up the environment, the service was free of charge. See `http://www.google.com/apps`

[3]For security reasons, we intentionally omit the URL of our identity provider service.

Model adjustment

As we said in Section 5.4, the protocol implementations may differ from the model due to HTTP 30x redirections. However, there are other differences that must be considered in order to execute tests. For example, both the SAML SSO and the OpenID specifications explicitly abstract away the user authentication, leaving to the implementation the choice of a particular mechanism. As a result, the counterexamples returned by the model checker do not specify how to pass the user authentication phase at the IdP. In order to support implementation-specific steps, we added the support for executing arbitrary user code at specific point of the protocol. This is done by adding a user-defined symbol into the model, and then by adding a mapping between the user-defined symbol and the code implementing the logic of the step.

SAML SSO We added `userlogin` into the `Client` and `IdentyProvider` entities between the step A2 and step A3 of Figure 3.1. For example, the `Client` has been modified as follows:

```
1   Actor -Ch_C2IdP-> IdP  : httpRequest (get, IdP, AReq,
2                                        nil_http_element);
3   userlogin;
4   IdP   -Ch_IdP2C-> Actor: httpResponse(code_200, nil_agent,
5                                        nil_http_element,
6                                        htmlForm(?SP, ?ARsp));
```

Listing 5.1: Modification of the `Client`

The `IdentyProvider` has been modified as follows:

```
1   C -Ch_C2IdP-> Actor   : httpRequest (get, Actor, httpBinding(
2                                authnRequest(?SP, Actor, ?ID), ?URI),
3                                nil_http_element);
4   if (TrustedSPs->contains(SP)) {
5     userlogin;
6     Actor -Ch_IdP2C-> C : httpResponse(code_200, nil_agent,
7                                nil_http_element,
8                                htmlForm(SP, postBinding(
9                                   signedAuthnResponse(inv(pk(Actor)),
10                                  SP, Actor, C, ID), URI)));
```

```
11    }
```

Listing 5.2: Modification of the `IdentyProvider`

OpenID We added `userlogin` into the client and OpenID provider entities between the step A3 and step A4 of Figure 3.5.

IUTs

We specified four IUTs, one for each implementation under test. For example, Listing 5.3 shows the mapping for testing SAML-based SSO for Google Apps.

```
1  <iut model="SAML_SSO-SP_init.aslan">
2   <map name="c" type="String" value="client"/>
3   <map name="code_200" type="Integer" value="200"/>
4   <map name="code_30x" type="Multiple-Value" value="302,301,303,304"/>
5   <map name="get" type="String" value="get"/>
6   <map name="i" type="String" value="intruder"/>
7   <map name="nil" type="Wild-Card" value=""/>
8   <map name="nil_agent" type="Wild-Card" value=""/>
9   <map name="nil_http_element" type="Wild-Card" value=""/>
10  <map name="post" type="String" value="post"/>
11  <map name="uri_i" type="URL" value="http://localhost:8081/resource"/>
12  <map name="uri_sp" type="URL" value="http://www.google.com/calendar/
        hosted/ai-lab.it"/>
13  <map name="userlogin" type="String" value="gmail.UserLogin"/>
14  <map name="htmlForm" type="Adapter" value="adapters.Html"/>
15  <map name="httpReq" type="Adapter" value="adapters.Http"/>
16  <map name="httpResp" type="Adapter" value="adapters.Http"/>
17  <map name="idp" type="URL" value="http://i/sso/IdP/process_response.
        php"/>
18  <map name="sp" type="URL" value="https://www.google.com/a/ai-lab.it/
        acs"/>
19  <map name="httpBinding" type="Adapter" value="adapters.Saml"/>
20  <map name="authnRequest" type="Adapter" value="adapters.Saml"/>
21 </iut>
```

Listing 5.3: Mapping for SAML SSO

Each entry of Listing 5.3 has a name, a type and a value. The name is an ASLan symbol. The type can be one of the following: Java string, Java integer, URL, adapter (i.e., Java class name), and multi-value. A multi-value type is a special type where all the values within the same multi-value

elements are equals. For example, the symbol `code_30x` is mapped with a multi-value whose values are 301, 302, 303, and 304. For generating a message, the constructors picks the first element of the list, while for parsing, the real values 301, 302, 303 and 304 are abstracted to `code_30x`.

It is worth noticing that the user-defined symbol `userlogin` is mapped with a Java class `gmail.UserLogin`. The class `gmail.UserLogin` implements the logic for executing the user login, e.g., submitting an HTTP form with the user name and the password of the user account used for our tests.

In our experiments, we tested the honest service provider *sp* and the identity provider *idp*. The client *c* and the attacker *i* are simulated by the instrumented model.

Test Case Execution

We run the prototype against the SAML-based SSO for Google Apps. The TEE automatically executed the attack traces till the message S2 of Figure 4.3 and, as expected, the message S2 contains the mailbox of the user. The tests against SimpleSAMLphp failed when message S2 was received. The analysis of the HTTP conversation has revealed that SimpleSAMLphp returns an error message instead of the message S2. We identified the cause of this error in a set of additional checks that were introduced in the code to 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`.

Finally, we run the attack trace of Figure 4.4 against Zoho Invoice relying party with Google OpenID identity provider and Yahoo OpenID identity provider. In both cases, the TEE reached the step S2, that, as expected, contains the resource requested in S1.

5.6 Conclusions

In this chapter we have shown that starting from the specifications of a protocol, it is possible to generate test cases by using a model checker and to automatically execute them against different protocol implementations. We developed a tool and used it to test the SAML-based SSO for Google Apps, the Foodle service, and the Zoho Invoice service against the flaw of Chapter 4. Our results show that the prototype is able to detect the logic flaw on the Google service and the Zoho service. Moreover, the prototype was not able to automatically confirm the authentication flaw on Foodle due to specific implementation mechanisms used by SimpleSAMLphp that mitigate the flaw.

The formal analysis of security protocols relies on the assumption that the specifications of a protocol are available. This assumption is still valid for the approach that we presented. However, the specification of a web application are almost never available in practice. Chapter 6 presents a technique for detecting logic flaws without the specifications of the system under test. The approach first infers a model from a number of HTTP conversations, then it generates and executes test cases that are likely to tamper with the application logic.

Chapter 6
Black-Box Detection of Logic Flaws

In the previous chapters, we saw how, starting from the formal specification of a protocol it is possible to automate the testing of real applications. However, specifications describing the evolution of the internal state and of the expected user behavior are almost never available for web applications. This lack of documentation makes it very hard to find logic vulnerabilities. In this chapter we propose a technique for detecting logic flaws when the specifications are not available. We applied our technique to seven large eCommerce applications executing more than 3100 test cases, 900 of which violated the expected behavior of the application. We discovered ten previously unknown logic flaws among which five of them allow an attacker to pay less or even shop for free.

Structure: This chapter is structured as follows. Section 6.1 gives an overview of the proposed approach. Section 6.2 and Section 6.3 present, respectively, the inference technique and the behavioral patterns. Section 6.4 presents the pattern-based test case generation, while Section 6.5 describes the test execution engine. Section 6.6 presents the test oracle that verifies for violation of the application logic. Section 6.7 discusses the setup for the experiments and Section 6.8 shows the results. Finally, Section 6.9 and

Section 6.10 respectively discuss the limitation of the approach, and draw the conclusions.

6.1 Overview

The OWASP Testing Guide v.3.0 [The08] suggests a 4-steps approach to test for business logic flaws in a black-box settings. First, the tester studies and understands the web application by playing with it and reading all the available documentation. Second, she prepares the information required to design the tests, including the *intended workflow* and the *data flow*. Then, she proceeds with the design of the test cases, e.g., by reordering steps or skip important operations. Finally, she sets up the testing environment by creating testing account, runs the tests, and verifies the results.

Our approach aims at automating the previous steps in a single black-box tool. First, starting from a list of network traces containing HTTP conversations, our system infers an application model and clusters resources related to the same workflow "step" (Section 6.2). Second, our technique analyzes the model and extracts a set of *behavioral patterns* (Section 6.3) modeling both the workflow and dataflow of the application. Third, we apply a set of *attack patterns* to automatically generate test cases (Section 6.4). Finally, we execute them against the web application (Section 6.5), and we use an *oracle* to verify whether the logic of the application has been violated (Section 6.6).

In the rest of the section we describe each phase in details using eCommerce web applications of Chapter 3 as a running example.

6.2 Model inference

The technique we present in this chapter is *passive* and *black-box*. This means that we do not require any access to the application source code (both on the client and on the server-side), and that we do not actively crawl the application pages or generate any traffic to probe its internal state. Instead, we take as input a list of traces containing sequences of HTTP requests and

responses. These traces can be manually generated by the tester, or just collected by logging real user interactions.

For simplicity, we consider only network traces that exercise a specific functionality of the web application. For example, if the web application under test is a shopping cart, we will use traces in which users log in, add items into the cart, and check out to buy the products. Nothing prevents the tester from generating traces that also contain other functionalities, such as browsing the online catalog or posting product reviews. However, focusing only on one aspect of the business logic helps our system to find the relevant operations with a minimum number of input traces.

Web applications often involve multiple parties. For instance eCommerce web applications typically involve the client, the eCommerce website, and the payment service. However, the communication between them is normally channeled through the user browser and, therefore, we focus on this point for the collection of our traces. In addition, it is useful to collect traces from different deployments of the same web applications, to allow our inference method to identify parameter values hard-coded in a certain installation.

Once the input traces have been collected, the first phase of our analysis consists of building the *navigation graph* of the application, enriched with the syntactic and semantic types for each parameter. The model inference is done in three steps: resource abstraction, resource clustering, and model refinement.

6.2.1 Resource abstraction

Input traces are sequences of pairs of HTTP requests and responses to request and fetch resources. Our approach currently supports the following resources: JSON data objects [Cro06] and HTML pages. However, it can be easily extended to other types such as SOAP messages [Wor07].

An HTML page is a resource that is displayed to the user within the web browser. It contains link and form tags that the user uses to request other resources. An HTML page can also contain client-side scripts such as JavaScript (JS) for generating AJAX requests for fetching asynchronously

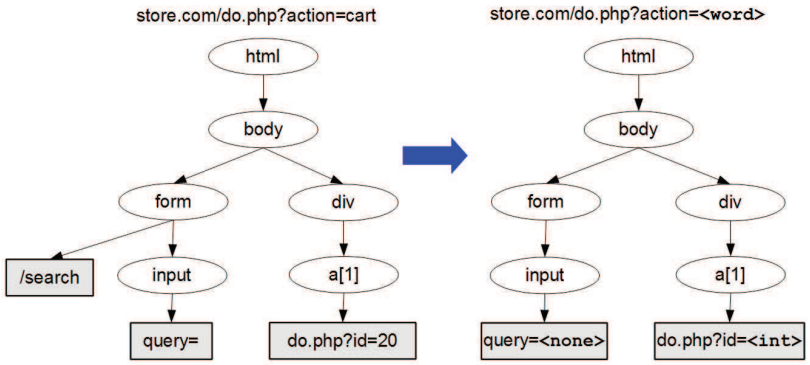


Figure 6.1: Resource abstraction and syntactic type inference of HTML page

other resources. As opposed to an HTML page, a JSON data object is not directly displayed to the user. It is an associative tree-like data structure that associates keys to data values. A JSON data object is requested asynchronously by client-side scripts and, once fetched, it is processed and inserted into the HTML page by using the DOM API [Wor05]. The JSON data object supports different data types such as numbers, strings, booleans, arrays, and objects [Cro06]. For example, the data values of a shopping cart can be sent to the web browser with the following JSON object:

```
{‘items’:
     {‘item1’: [’price’: 19.9,
                ‘tax’  :  1.6],
      ‘item2’: [ ... ]
     }
}
```

The example above shows five keys: ’items’, ’item1’, ’item2’, ’price’, and ’tax’. Each key is associated to data types. The keys ’price’ and ’tax’ are associated to data values of type number. The key ’item1’ and ’item2’ are associated to one array each. Finally, the key ’items’ is associated to a data value of type object.

At this step of the inference, we aim at extracting from the resources information related to the dataflow and the workflow of the web application.

The information related to the dataflow are the data values exchanged between the web application and the web browser. These data values are placed by the web application inside the HTML page in the form of HTML forms, HTML links, or inside the JSON data objects. For example, the left-hand side of Figure 6.1 and Figure 6.2 shows the tree-representation of an HTML page and a JSON data object. The leaves are the URLs of the input fields carrying data values that may be used to submit new HTTP requests to the web application. The leaves as well as the URL, the POST data, and HTTP redirections are used in Section 6.3 to extract the data values propagation patterns.

As opposed to the dataflow, the information specific to the workflow cannot be directly identified in HTML tags or JSON keys. In our approach, we extract the workflow information by first clustering together resources that share similarities, and then by inferring behavioral patterns such as the occurrences in the trace. The clustering is detailed in Section 6.2.2 and the behavioral patterns are explained in Section 6.3. At this stage of the inference technique, the resource abstraction extracts from the HTML code and JSON data object the DOM path of HTML tags that would allow a comparison between the resources. With reference to Figure 6.1 and Figure 6.2, the paths from the roots till the leaves are the position of the leaves inside the resource. The comparison between JSON data object, and in a similar way between HTML pages, is done via the comparison of the paths.

We call *abstract HTML page* the collection of (i) its URL, (ii) the POST data (iii) the anchors and forms contained in the HTML code and their DOM paths, (iv) the URL in the meta refresh tag, and (v) the HTTP redirection location header.

Figure 6.1 shows the tree-representation of the DOM paths of all the anchor and form elements. We treat HTTP redirections as special HTML resources.

We call *abstract JSON object* a collection of (i) its URL, (ii) the POST data, (iii) the pairs of value and path in the object and (iv) the HTML links if any HTML code is contained.

From each abstract resource we extract a set of elements corresponding

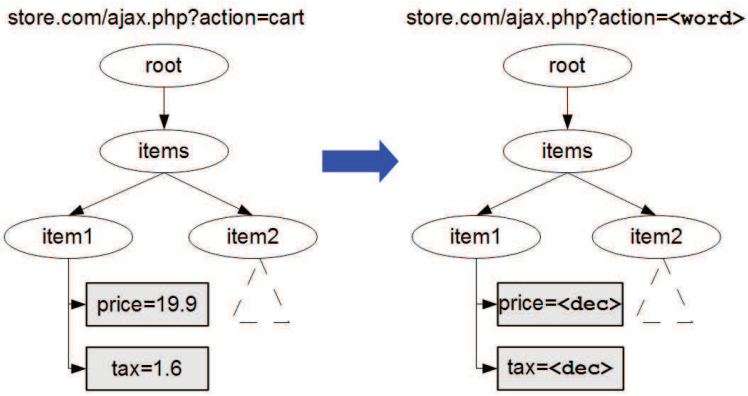


Figure 6.2: Resource abstraction and syntactic type inference of a JSON data object

to all possible parameters that appear in the URLs, in the POST data, and in all the links. Each *element* is characterized by a name, a value, a path, and an inferred syntactic type. Our approach supports the integer type, decimal type, URL type, email address type, word type (alphabetical strings e.g. "add", "remove", ...), string type, list type (comma-separated values), and *unknown* type for everything else. The type is automatically associated to each element by inspecting the values that the element had in the input traces. Obvious priority rules are applied in case of ambiguity – e.g. id=20 can be both a number and a string, but being the first a subset of the second, it is considered to be a number.

6.2.2 Resource clustering

Modern web applications map application logic operations to different resources. For instance, the operation of displaying the shopping cart could involve an initial HTML page containing the skeleton of the web page and then a collection of AJAX requests for populating the page with the list of items, tax, available vouchers, and so on. Given a list of network traces we would like to cluster the resources when they are likely to encode the same operation of the application logic. We cluster resources in three phases. First, we relate AJAX requests to the resource that originated them. Then

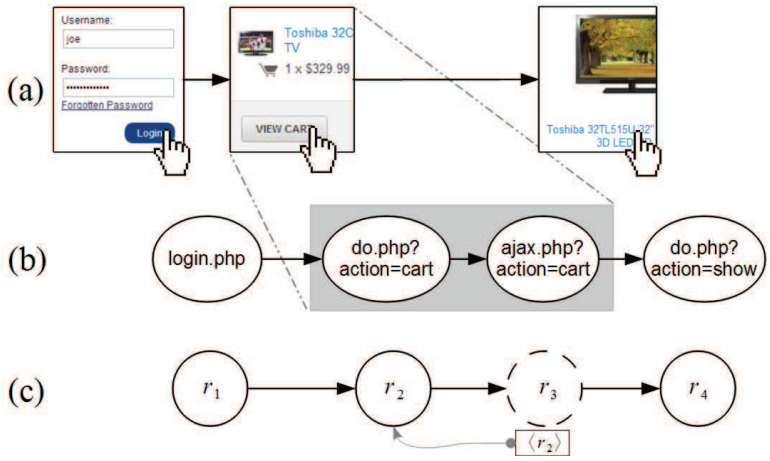


Figure 6.3: (a) Application-level actions, (b) URLs requested, and (c) abstract resources with list of originators

we group together resources considering their similarity and the originators. Third, we split a cluster if a parameter of its resources encodes a command rather than carrying a value.

During the first phase, we preprocess input traces to identify AJAX requests. This can be done by checking the "X-Requested-With" HTTP request header [The13a] or by detecting JSON responses. After that, we associate each resource to its originators. Figure 6.3 provides an example of this first phase. In Figure 6.3.a we have a segment of input trace in which the user logs in, checks the status of the shopping cart, and finally accesses the details of a product. These application-level actions are mapped to the resources shown in Figure 6.3.b, that are abstracted in Figure 6.3.c. In Figure 6.3.c we have the HTML page r_1 followed by the page r_2. Then, r_2 requests r_3 by using AJAX that enriches r_2 with new HTML code, or new client-side scripts. The example then ends with r_4 that we assume to be caused by a link in r_2 or added by r_3. Figure 6.3.c shows also the list of originators of each resource. r_1, r_2, and r_4 have no originators, while r_3 was originated by r_2.

In the second phase, we cluster resources. In general, two resources are in the same cluster if they have the same URL, the same POST parameters,

and, if any, the same redirection URL. When comparing URLs we do not take into account the values of the parameters, but only their syntactic types. For example, the following three URLs are equivalent:

```
store.com/do.php?action=add&id=3
store.com/do.php?action=add&id=7
store.com/do.php?action=show&id=3
```

We compare first synchronous resources as explained before. For instance, the resources r_2 and r_4 of Figure 6.3 are in the same cluster. Then, we compare asynchronous resources. Two asynchronous resource are in the same cluster if they have the same URL, POST data, redirection URL, and originators. For instance, let us suppose to compare r_3 with another asynchronous resource r' (not shown in Figure 6.3). After comparing the URLs and POST data, we compare whether the originators of and r' are in the same cluster.

During the last phase, we visit each cluster and we try to identify the parameters that are encoding a command rather than just transporting a value. For each parameter we take the sub-group in which pages have the same value for that parameter. For example, the parameter `action` divides the gray cluster of Figure 6.4.a in two sub-groups, one for the `cart` value and one for the `show` value. We then compute the *page similarity* between pages in the same sub-group and between pages in different sub-groups. The comparison is done by looking at the DOM path of HTML forms, their action attribute (URL domain and parameter names), and the name of the nested input elements. The function is applied to sub-groups by calculating the percentage of pages that are similar. If the similarity inside the same sub-groups is high (more than 55%), and between different sub-groups is low (less than 45%), then we assume the parameter is used to specify an operation and we create a different node for each value. Otherwise we leave the cluster unmodified. The result of this phase is shown in Figure 6.4.b.

Once we have grouped all the resources in clusters, we can build the navigation graph. The navigation graph is a directed graph $G = (\mathcal{C} \cup \{I, F\}, E)$ where $\mathcal{C}$ the set of clusters, I the source node, F the final node, and E the

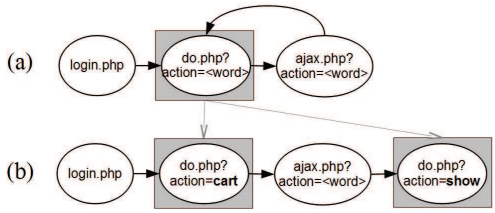


Figure 6.4: (a) Clusters after comparing all the resources (b) Clusters after having identified parameters encoding a command

set of edges initially empty. We place the edge (u, v) if there exists one input trace π in which a resource $r' \in u$ immediately precedes a resource $r'' \in v$. Then, for each r_j at the beginning of each trace (i.e. $\pi = \langle r_j, \ldots \rangle$), we place the edge (i, u) where $r_j \in u$ and for each r_j at the end of each trace, (i.e. $\pi = \langle \ldots, r_j \rangle$) we place the edge (u, f) where $r_j \in u$. Finally, we associate to each node u the set of all the elements for every $r \in u$.

Model Refinement

In the final step of the model inference, elements associated to nodes are enriched with semantic types. In [WCWQ11, WCW12] Wang et al. proposed to label URL parameters with syntactic and semantic attributes. Our work borrows few of their types (Server- and Client- generated attributes) and add new ones. The list of semantic types supported is the following:

Unique type - for parameters whose values is different in each page within the same node

Constant type - for parameters that have always the same value in all resources within the same node

Server-generated type - for parameters whose values appear in HTTP responses before appearing in any HTTP request

Client-generated type - for parameters whose values appear in an HTTP request before appearing in any HTTP response.

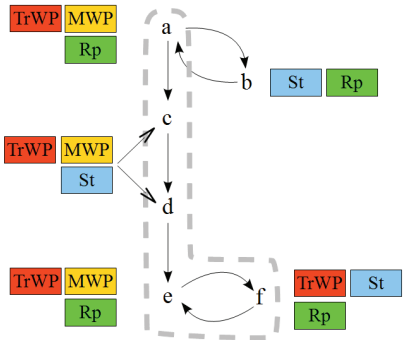


Figure 6.5: Example of behavioral patterns using $\pi_1 = \langle a, b, a, c, d, e, f, e \rangle$ and $\pi_2 = \langle a, c, d, e, f, e \rangle$

Intuitively, the first two types describe properties of parameters that are true inside a node, while the last two describe properties of parameters that are true in the same input trace.

6.3 Behavioral Patterns

During the second phase of our approach, we analyze the navigation graph and the input traces looking for patterns that are likely related to the underlying application logic. We divide workflow patterns in *Execution Patterns*, that model what users normally do in our input traces, and *Model Patterns* that model what the navigation graph allows to be done. Finally, *Data Propagation Patterns* model how data is propagated throughout the navigation graph.

6.3.1 Execution Patterns

Execution patterns model the actions performed by the user in the input traces. In particular, we focus on three patterns:

Singleton Nodes

A node is a singleton if it is never visited more than once by any input trace. Some of the users may visit these nodes, and some may not - but no one visit them twice. For example, submitting a discount voucher

can be an operation observed in some of input traces but none of them is submitting a voucher twice.

Multi-Step Operations

A Multi-Step Operation is a sequence of nodes always visited in the same order. This is very common in many functionalities in web applications. For example payment procedures or user registrations often consist of precise sequences of steps, and all traces going through those processes always execute them in the same exact order.

Trace Waypoints

We use the concept of waypoints to identify nodes that play an important role in the interaction between the user and the web application. In particular, trace waypoints are those nodes that appear in all the input traces. For example, if all our traces contains a purchase, then the redirection to the payment website (e.g., PayPal) is a trace waypoint.

6.3.2 Model Patterns

Model patterns model the sequences of actions that are allowed according to the navigation graph:

Repeatable Operations

Nodes that are part of a loop in the navigation graph are operations that can potentially be repeated multiple times.

Model Waypoints

Model waypoints are those nodes that belong to every paths in the navigation graph that go from the source node to the final node. In other words, these nodes are not only visited in all input traces, but there is no way in the navigation graph to bypass them. Therefore, every model waypoint is by definition also a trace waypoint but not vice versa.

Figure 6.5 shows a little example to better describe the difference between model and execution patterns. The example shows the behavioral patterns of

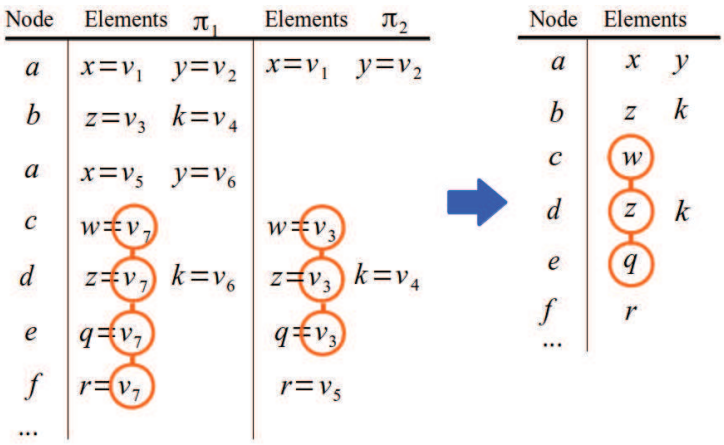


Figure 6.6: Propagation Chains: from traces to the navigation graph

a navigation graph extracted from two input traces $\pi_1 = \langle a, b, a, c, d, e, f, e\rangle$ and $\pi_2 = \langle a, c, d, e, f, e\rangle$. The symbols St, TrWP, Rp, and MWP stand for, respectively, singleton nodes, trace waypoints, repeatable nodes, and model waypoints. The thick dotted line delimits an example of multi-step operation. The way in which these patterns are combined together to test the web application is presented in Section 6.4.

6.3.3 Data Propagation Patterns

Propagation chains identify those cases in which the same variable is sent back and forth between the client and the web application. Our approach uses the propagation chains in two phases. First, it uses them during the test case generation in order to replay values of propagation chains across different user sessions. Second, it uses the propagation chains during the test execution in order to fetch the data value to submit it to the web application.

From an operational point of view, a propagation chain is a set of elements associated to nodes that have the same values. Two parameters in the model carry the same value if there are some input traces in which they hold the same value, and there are no traces in which the values are different (since the user does not perform the same actions in all the traces, a certain parameter may not be present in all of them).

We compute this in two steps. First we identify the propagation chain of each value within a trace. For example let us consider the example in Figure 6.6. Here, in the input trace π_1, the element w of the node c has the same value of the elements z, q, and r of the nodes d, e, and f respectively. Looking at trace π_2, the elements w of the node c is still equal to the parameters z and q respectively in d and e, but it is now different from q.

By comparing these relationships, we obtain the propagation chain depicted in the right side of Figure 6.6. A value propagates between two nodes, linking together the variables w and z. Note that the relationship between r and q in π_1 has been invalidated by π_2 and therefore is not included in the final model.

We say that the chain is *client generated* if the initial value is chosen by the user, and *server generated* otherwise. A similar classification is used by Wang et al. [WCW12]. However, their notion is limited to input traces of the same length while ours is extended to traces of different lengths and to the application models.

6.4 Test Case Generation

In this section we describe how we generate test cases to stress the logic of the web application. This is done by using attack patterns that simulate an attacker that tries to use the application in an unconventional way. In particular, we focused on a set of actions an attacker could perform: repeating operations, skipping operations, subverting the order of operations, and mixing parameter values across user sessions. For each action we designed a pattern. An example of these attack patterns is presented in Figure 6.7. These examples are based on the navigation graph of Figure 6.5. We enriched the graphs with numbers for showing the order in which the nodes are visited. For simplicity, we are omitting the source node I and the final node F, respectively connected to a and e.

Figure 6.7: Test case generation patterns

6.4.1 Multiple execution of repeatable singletons

This attack pattern is obtained by comparing execution patterns with model patterns. If the model identifies a node as "repeatable" but the traces mark it as "singleton", it means that even though there seems to be a way to execute an operation multiple times, this was never observed in our normal training set. Therefore, it may be interesting to see what happens when these operation are repeated.

Figure 6.7.a shows the sequence of steps in the test case. If b is a repeatable singleton node, we select an input trace that visits b (e.g. $\langle a, b, a, c, d, e, f, e\rangle$). This path is then cut into two parts at the node before the singleton, that is $\langle a\rangle$ and $\langle b, a, c, d, e\rangle$. We call these two parts *test case prefix* and *suffix*. Second, we compute the shortest loop on the navigation graph that brings from the singleton node to itself, that is $\langle b, a\rangle$ in our example. Finally, the test case is obtained by concatenating the prefix, the loop, and the suffix paths.

6.4.2 Breaking Multi-Steps Operations

There are several ways of subverting the order of multi-steps operations. We distinguish two types. The first approach is to give a different order to the steps composing the operation. For example, let us suppose that the nodes a, c, d, e, and f in Figure 6.7.b compose a multi-steps operation in the given order, we can try to execute the sequence as $\langle a, d, c, e, f\rangle$. The second approach is to break the sequence by interleaving one of the already performed operations. For example, between d and e we could go back and visit again c (e.g., executing $\langle a, c, d, c, e, f\rangle$). For instance, after adding the taxes to a checkout process, we could go back and change the number of items in the shopping cart. Our approach currently support the latter type, however it could be extended to support the former type or other types of reordering.

6.4.3 Breaking server-generated propagation chains

The goal of this attack pattern is to execute two user sessions and then replace a value generated by the server in one session with the one of the other session. As shown in Figure 6.7.c, the corresponding test case is composed of two parts. The first part has the goal of interacting with the application and capturing the value of a server generated propagation chain. The second part starts another session and interrupts the propagation chain replacing the value of `z` with the value v previously captured.

This attack pattern can potentially generate a very large number of test cases. In fact, each web application can contains a lot of identifiers generated by the server (for instance, all the product or message IDs). Therefore, we focus our test case generation only on two types of propagation chains: the ones containing unique values (i.e., that differ in all the input traces and are therefore related to the session) and the ones containing installation-specific values (i.e., values that are constant only within the same installation).

The test case generation algorithm operates as follows. First, we select the parameters belonging to the chain that appear inside an HTTP request. These parameters are called *injection points* and model the point in which an attacker can replace the value generated by the server. For example, in Figure 6.7.c the parameter `z` of the node d is an injection point. Second, we select two traces from different user sessions that are visiting the node associated to the injection points. The first is truncated at the injection point and the second, unmodified path, is appended to the first one. With reference to Figure 6.7.c, the two paths are respectively at the left- and right-hand side.

In general, our approach only requires input traces that exercise a business function of the web application. However, depending on the type of attack pattern used, the input traces should satisfy further requirements. For example, in order to apply this attack patterns, the tester must provide input traces from different users and, optionally, from different web application installations.

6.4.4 Waypoints Detour

Waypoints are operations that are executed always by all the input traces. When these operations happen only once per input trace, they seem to indicate some sort of milestone in the execution of the business process of the web application. In this case, the attacker can try to skip one or more of these operations.

We consider two attack patterns that skip waypoints. The first is shown in Figure 6.7.d in which the attacker skips an individual waypoint, e.g. d. The second is depicted in Figure 6.7.f in which the attacker skips two operations, e.g. $\langle c, d \rangle$.

The generation of test cases following the pattern in Figure 6.7.d is straight forward. First, we select an input trace that visits the waypoint d, e.g., $\langle a, b, a, d, e, f, e \rangle$ and then we remove the node d. The result is $\langle a, b, a, e, f, e \rangle$.

The generation of test cases for the attack pattern in Figure 6.7.e requires further care. In this case, we consider also the interference of the attack pattern to the propagation chain. For example, it may happen that skipping a waypoint can interrupt a propagation chain of a data value that is needed for the subsequent actions. For example, let us suppose that the URL of the node e requires the parameter q whose value appear in the resource d. In this case, we use a similar approach as seen in breaking of server-generated propagation chain in which the values of q is taken from another user session.

The generation of the test case is the following. First, we select a pair of waypoints c and d. Then, we select two input traces from two users. The first input trace is truncated at the first occurrence of e, i.e., $\langle a, b, a, c, d, e \rangle$. The second skips the steps c and d, i.e., $\langle a, b, a, e, f, e \rangle$. The result is the following test case $\langle a, b, a, c, d, e \rangle.\langle a, b, a, e, f, e \rangle$.

6.5 Test Case Execution

The test cases described in Section 6.4 are abstract representations that still miss details required to be properly executed. For example, the values

of some parameters cannot be determined from the model and needs to be collected during the test case execution. The execution engine has to treat different parameters in different ways, taking the constant from the models, others from the values observed in the input traces, and preserving the propagation of the ones that are generated by the application at runtime.

In addition, it is important that after each test the application is reset to its initial state to avoid interferences between consecutive executions. For example, a test may leave a number of items in the shopping cart, thus affecting every following experiment performed with the same user account. In general, it is often enough to delete the cookies and empty the shopping cart at the end of each test.

The execution engine iterates over each page in the test case and turns them into an abstract HTTP request. Constant values and propagation chains are then assigned to the request parameters to generate a concrete request that is executed by the engine. The response is parsed in order to extract server generated parameters and update their current values. If the execution engine is not able to properly reconstruct a chain (e.g., because the page that was supposed to generate its value returned an error) the execution engine abort the execution and report the exception.

We say that a test is *correctly executed*, if the test execution engine ends with no exception. Otherwise, we say that it is *not correctly executed*.

6.6 Test Oracle

The approach we propose in this chapter is completely independent from the business logic of the web application. Our technique can automatically identify behavioral patterns, and then generate test cases to break those patterns in a number of different ways. The system can also determine if a given test was executed correctly, but this is as far as it is possible to go with an application-agnostic approach. For example, replacing the value of a security token in a payment workflow would probably make the entire process fail. Unfortunately, without any knowledge about the underlying business logic, the test verdict could only say whether the pattern was applied suc-

cessfully, but it can not draw any conclusion about the possible implications. Therefore, if we want our tool to be able to report possible violations of the application logic, we need to extract the sequence of events that occur during a test case execution and compare them with the *logic property* that we want to violate.

A simple way to express a logic property for shopping carts could be the following: *if an order is approved for a user, then the user must have completed a payment for the corresponding amount.* In this formulation two events play a central role: the fact that an order is placed, and the fact that a user has paid a certain amount. Another important aspect of this property is the time dependency between the two events. Since propositional logic can only express truth regardless of the time, in our approach, we model logic properties as Linear Temporal Logic (LTL) formulas [Pnu77, Hol04]. LTL adds temporal connectives like $\mathtt{O}$ (once in the past) to traditional logical operators like $\wedge$ (and), and $\implies$ (implies). This enables to verify whether one event will eventually happen in the future or it already happened in the past. For example, the above logic property can be written in an LTL formula as follows:

$$ord_{placed} \wedge onStore(S) \implies \mathtt{O}(paid(U, I)) \tag{6.1}$$

where ord_{placed}, $onStore(S)$, and $paid(U, I)$ are respectively the events *order placed*, *operation performed on the store* S, and *user* U *paid the price of item* I. Now, the problem of identifying violation of the logic property is recast into the problem of checking whether the LTL formula is satisfied or not by a given test case.

In our approach, the *Test Oracle* is the component that given an execution of a test case returns *true* if a certain predefined logic property is violated, and *false* otherwise. The oracle is composed of two parts: an *events extractor* and an LTL formula checker. The extractor collects from the executed test a partially ordered set of events (events can happen in sequence or in parallel) grouped by user sessions. The second part verifies whether all

sequences satisfies or not the provided LTL formula.

It is important to note that both the events and the LTL formula depend on the type of applications under test and on the type of vulnerabilities that we are interested to find. For example, to find authentication bypass vulnerabilities it would be interesting to observe events related to the user login and to the access of private pages. However, since in this chapter we focus on the test of eCommerce applications, we are more interested in monitoring the money transfer and the value of the purchased items, as described in more details in the next section.

6.7 Experiments

In this section we describe the experiments we performed on a number of popular shopping cart applications. We first introduce the web applications under test, then we discuss the extracted models and the results of our automated analysis.

6.7.1 Shopping carts

We tested our technique on a number of popular shopping carts available for offline testing. Our tool can be used also to test online eCommerce applications (such as the Amazon store). However, our tests require to attempt malformed operations and to complete a large number of checkout processes. This would be both unethical, since the application can malfunction as a result of our tests, and very expensive, since it would have required to buy at least one product for each test case. Therefore, we opted for seven well known open source applications, as reported in Table 6.1. The table also shows the applications popularity measured according to the search results obtained by performing a number of *googledorks* [Hac13]. Each Google query was built by combining both the URL structure (e.g., the path of one of the cart's operation) and some static HTML content extracted from the application's pages (e.g., the "powered by..." text in the footer). As such, the numbers reported in the table are only a lower bound of the number of

Table 6.1: Popularity index

WebApp	Installations	WebApp	Installations
OpenCart	9,710,000	TomatoCart	119,000
Magento	3,130,000	osCommerce	80,500
PrestaShop	650,000	AbanteCart	21,200
CS-Cart	260,000		
		Total	13,970,700

publicly-accessible installations available on the Internet.

This conservative measurement shows that these seven applications are used by almost 14 million eCommerce installations. As a comparison, the two applications tested by Wang et al. [WCW12] returned less than 40,000 hits using similar Google dorks.

General setup

We installed two instances of each web application on our own servers. We will refer to them as the Store A and Store B. All installations except for AbanteCart and PrestaShop were then configured to use both the PayPal Express Checkout [Pay12a] and the PayPal Payments Standard [Pay12b] methods. In total we obtained 12 different configurations to test[1]. We left the other configuration options to the default ones.

We configured all the e-shopping applications in *SandBox* mode. In this configuration, each application performs transactions by using the PayPal SandBox payment gateway. These payments do not involve real money as they are performed between the seller and buyer testing accounts.

To generate the input traces we created two user accounts, U_1 and U_2, each controlling a PayPal buyer testing account. For each web application we captured in total six HTTP conversations, three for each store: one with U_1 buying one item, one with U_2 buying another item, and one with U_1 buying two different items. These input traces were sufficient to stimulate

[1]When we did the experiments AbanteCart and PrestaShop were providing only one of the two payment flows above, respectively PayPal Payments Standard for AbanteCart and PayPal Express Checkout for PrestaShop.

the main shopping cart functionality, but a better training could be used in the future to expose also more subtle features or configurations.

6.7.2 Testing Oracle

In their experiments, Wang et al. [WCWQ11] used the following logic property to describe shopping cart applications:

"The store S changes the status of an item I to "paid" with regard to a purchase being made by user U if and only if (i) S owns I; (ii) a payment is guaranteed to be transferred from an account of U to that of S in the CaaS; (iii) the payment is for the purchase of I, and is valid for only one piece of I; (iv) the amount of this payment is equal to the price of I."

However, this property is not entirely verifiable in a black-box setting. For instance, it is not possible to test the truth of the predicate "S owns I" nor to check whether the due amount has been transferred to the merchant's account. According to that, we simplified the above invariant by removing the non-verifiable clauses. The new property that can be used for automated black-box testing becomes:

When the store S confirms the user U that an order has been placed, then in the past U paid S the amount equal to the price of I and U agreed on purchasing I from S.

We modeled the invariant using the following events extracted during each test case execution:

- ord_{placed} when the shop confirms that the order has been placed;
- $onStore(S)$ when an operation has been performed on the store S;
- $paid(U, I)$ when the user U authorizes the payment gateway to pay the price of I;
- $toStore(S)$ when the payment is meant for the store S;
- $ack(I)$, when the user acknowledges to buy I.

These events are combined together in the following LTL formula modeling the logic property of the shopping cart applications:

$$\begin{aligned} & ord_{placed} \wedge onStore(S) \implies \\ & \quad \mathtt{O}(paid(U,I) \wedge toStore(S) \ \wedge \\ & \qquad \mathtt{O}(ack(U,I) \wedge onStore(S))) \end{aligned} \tag{6.2}$$

6.7.3 Test Case Execution

By applying our attack patterns to the models extracted from the input traces, we generated around 3100 test cases, an average of 262 per application. The number of test generated in each category is summarized in Table 6.2.

Table 6.2 also shows the portion of test cases that were successfully executed. An execution failed when, by applying one of the attack template, the resulting test case brought the application in a state in which it was impossible to proceed with the rest of the test (e.g., because of an error page was returned in an intermediate step). This is a common result, since by definition our tests try to stress the application to expose some unexpected behavior. The number of test cases violating our LTL formula is reported in Table 6.3. As mentioned before, there are events that are not visible to the oracle. Therefore, a violation to the LTL formula does not always correspond to a vulnerability. In fact, it is possible that further checks performed in the back end of the application would detect the fraud and cancel the order. In order to distinguish real vulnerabilities from other forms of bugs (e.g. erroneously reporting to the user a failed transaction as successful) we manually inspected the balance sheets of the merchant, the list of orders, and their status. Whenever the result was not confirmed by our manual inspection, we discarded it as a non-vulnerable case. The remaining cases correspond instead to anomalous behaviors associated to real software vulnerabilities, as explained in the next Section. It is important to note that over 28.9% of the test cases generated by our approach brought the application in a state that violated our oracle test, and 1 test out of 52 exposed a previously unknown

Table 6.2: Test case generation and execution

		Generation					Execution			
WebApp		Time	Repeat	Detour	MSteps	PChains	Time	Exec.	Not Exec.	Total
AbanteCart	Std	$\ll$ 0:01	9	152	51	21	4:51	74	159	233
Magento	Exp	0:02	10	246	82	5	16:23	240	103	343
	Std	0:02	14	303	62	7	14:50	210	176	386
OpenCart	Exp	0:01	10	83	77	3	2:34	140	33	173
	Std	0:01	15	60	38	22	2:08	71	64	135
osCommerce	Exp	$\ll$ 0:01	4	142	13	6	3:22	117	48	165
	Std	0:01	8	144	63	10	3:42	128	97	225
PrestaShop	Exp	$\ll$ 0:01	12	100	22	3	2:42	85	52	137
TomatoCart	Exp	0:02	9	215	68	10	4:54	238	64	302
	Std	0:02	17	138	32	37	4:36	115	109	224
CS-Cart	Exp	0:05	8	562	24	6	12:02	347	253	600
	Std	0:02	16	137	54	15	5:29	127	95	222
Total			132	2282	586	145		1892	1253	3145

Table 6.3: Results

WebApp		Viol.	Bugs	Vuln.
AbanteCart	Std	17	16	1
Magento	Exp	65	65	-
	Std	126	126	-
OpenCart	Exp	58	46	12
	Std	30	30	-
osCommerce	Exp	42	22	20
	Std	35	34	1
PrestaShop	Exp	-	-	-
TomatoCart	Exp	90	65	25
	Std	24	24	-
CS-Cart	Exp	313	313	-
	Std	109	108	1
Total		909	849	60
		100%	93.4%	6.6%

logic vulnerability.

Test case generation does not require much resources, while the execution phase can be quite time consuming (max 16h for the Magento). This is largely due to the lack of parallelization in our experiments, and to the fact that the PayPal sandbox environment is much slower than its live counterpart. The model inference – omitted from Table 6.2 – required an average of 9 minutes per application for building the navigation graphs that, in average, contained 34 nodes and 48 edges.

6.8 Results

In this section we discuss the results of our experiments. 6.6% of the problems identified by our tests correspond to vulnerabilities. They were confirmed manually by inspecting the merchant/buyer balance sheets in the merchant PayPal account, and the status of the orders available in the back-office of the online store. The remaining 93.4% are bugs in the presentation in the eCommerce application. In these cases, the tests were executed until the final page in which the store congratulates the customer for the purchase.

This caused the generation of the events $ord_{placed} \wedge onStore(S)$. However, a manual inspection revealed that all the orders in the database were "not present", "not complete", or "unpaid".

6.8.1 Vulnerabilities

Table 6.3 shows that 60 of our test cases (1.9% of the total) exposed a logic vulnerability in the target applications. We discovered the following flaws:

- In osCommerce v.2.3.1, CS-Cart v.3.0.4, and AbanteCart v.1.0.4 with PayPal Payments Standard a malicious customer can shop for free (exploitable)
- In OpenCart v.1.5.3.1 and TomatoCart v.1.1.7 with PayPal Express Checkout a malicious customer can pay less (exploitable)
- In TomatoCart v.1.1.7 with PayPal Express Checkout a malicious customer can shop for free (exploitable)
- OpenCart v.1.5.3.1, TomatoCart v.1.1.7 and osCommerce v.2.3.1 with PayPal Express Checkout a customer can pay an amount different from what she authorized (not exploitable)
- TomatoCart v.1.1.7 with PayPal Express Checkout a customer pays another customer's cart (not exploitable)

All the exploitable flaws have been already responsibly disclosed. When the developers did not answered within two weeks of our notification, we reported the vulnerabilities also to the US Cert[2]. In the following we describe each class of vulnerability we discovered in our experiments.

osCommerce, CS-Cart, and AbanteCart with PayPal Payments Standard - shopping for free

These flaws have been discovered by test cases that interrupted the server-generated propagation chain transporting the PayPal account of the mer-

[2] `http://www.kb.cert.org/vuls/id/459446`, `http://www.kb.cert.org/vuls/id/207540`, `http://www.kb.cert.org/vuls/id/583564`

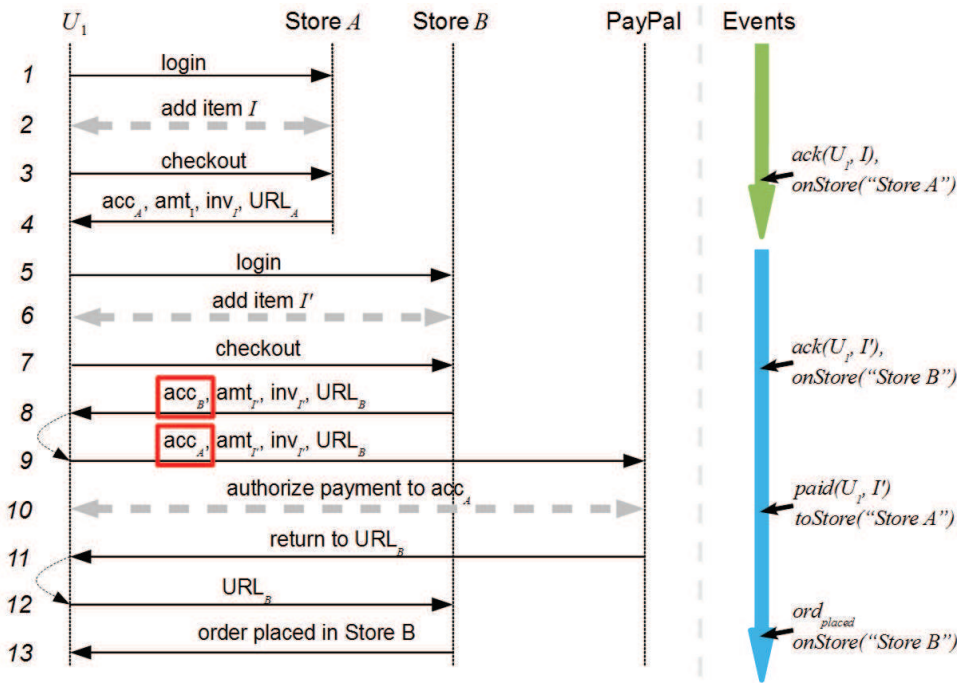


Figure 6.8: Shopping for free with osCommerce v.2.3.1 and AbanteCart v.1.0.4

chant.

An example is shown in Figure 6.8. The left-hand side of the Figure shows the message sequence chart while the right-hand side shows events grouped by user session. Each user session begins with a *login* message. The events show how the violation was detected. At the end of the execution, the clause $ord_{placed} \wedge onStore(\text{“}Store\ B\text{”})$ is satisfied as all the events in it were observed. However, the formula $\mathtt{O}(paid(U, I) \wedge toStore(\text{“}Store\ B\text{”}) \wedge \mathtt{O}(ack(U, I) \wedge onStore(\text{“}Store\ B\text{”})))$ is not satisfied because none of the events in it were observed in the past.

The manual inspection verified that (i) no payment was made to the Store B, (ii) the status of the order in the back office of Store B was marked as “completed”, and (iii) the invoice resulted paid.

It is straightforward to turn the above test case into a real attack. Indeed, when redirected to PayPal, an attacker can replace the seller PayPal account

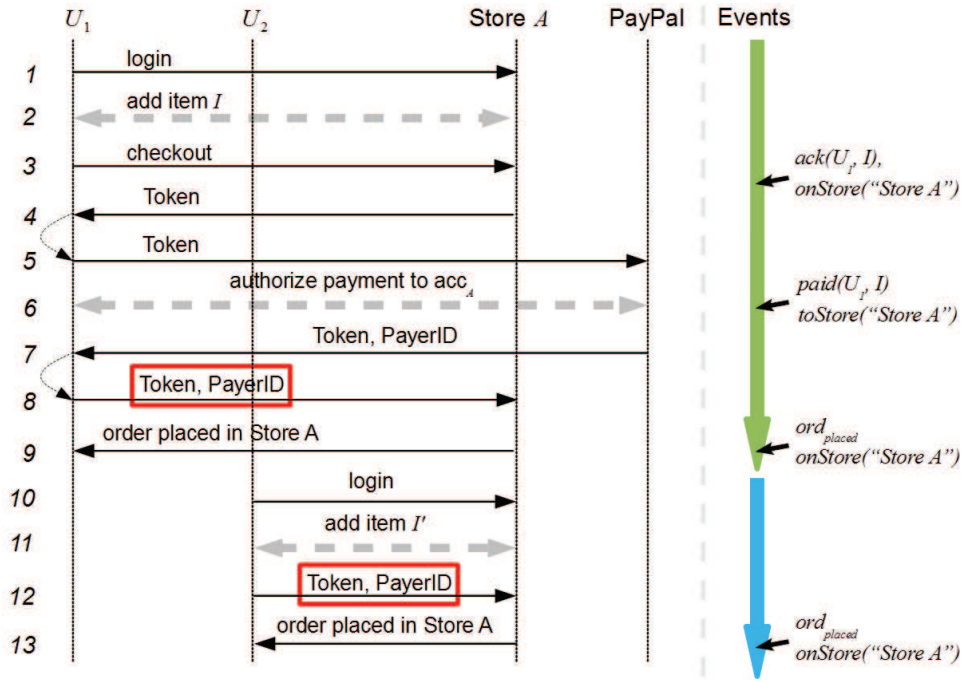


Figure 6.9: Paying less with OpenCart v.1.5.3.1 and TomatoCart v.1.1.7

with another PayPal account the attacker controls. This results in placing an order and by paying herself instead of the real shop.

OpenCart and TomatoCart with PayPal Express Checkout - pay less

In OpenCart and TomatoCart with PayPal Express Checkout an attacker can pay less than the value of the goods she is purchasing. The flaw has been detected by using the waypoints detour pattern. The test case generator produced 11 test cases for OpenCart and 11 for TomatoCart in which the user U_2 skips the nodes of the redirection to PayPal for the payment and reconstructs the URL with values taken from the user session of U_2. A representative test case is shown in Figure 6.9.

The events in Figure 6.9 shows that during the second user session the clause $ord_{placed} \wedge onStore(\text{“}Store\ A\text{”})$ is satisfied. However, the other clauses

of the formula are not satisfied because neither the user acknowledgment nor the payment were observed.

The manual inspection found two distinct orders in the list of orders, one for I and for I'. Both orders were in the state "paid" and ready for shipping. However, the balance sheet of the merchant contains only the transaction for I, while nothing is recorded for I'.

This test case can be turned into an attack by first buying a cheap item and intercept the redirection URL from PayPal to the store. Then, the attacker can login again, add an expensive item to the cart and replay the URL captured before. The store responds with a confirmation page. Even worse, we verified that the attacker (or any other user) can reuse the same $TokenID$ and $PayerID$ to complete an arbitrary number of additional fake transactions. This process is only bounded by the timeout set by PayPal on the token.

TomatoCart with PayPal Express Checkout - shopping for free

This problem has been identified by 11 different test cases generated with the waypoint detour pattern. A representative test case is shown in Figure 6.10. The events in Figure 6.10 shows that during the second user session the clause $ord_{placed} \wedge onStore(\text{"}Store\ A\text{"})$ is satisfied. However, the other clauses of the formula are not satisfied because neither user acknowledgment nor the payment were observed.

The manual inspection verified that no payment for I and for I' were done. However, the list of orders contained the order for I' in a "paid" state and ready for shipping.

This test case can be turned into an attack as shown before with the difference that the attacker ends the first user session at step 7.

osCommerce, OpenCart and TomatoCart with PayPal Express Checkout - pay less

In osCommerce the test was generated by the waypoints detour, while in OpenCart and TomatoCart the discovery was done by breaking server-generated

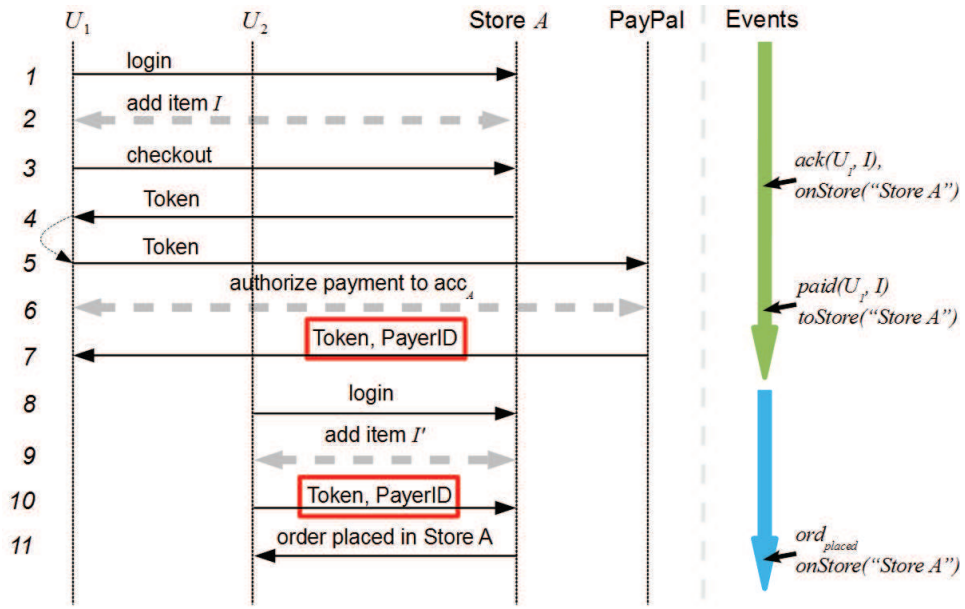


Figure 6.10: Shopping for free with TomatoCart v.1.1.7

propagation chains.

For osCommerce, the test case is similar to the one shown in Figure 6.10. The events show that the order made by U_2 was placed but no payment was observed. For OpenCart and TomatoCart, the tests are similar to the one in Figure 6.8. However, the difference is the chain that is interrupted. When PayPal Express Checkout is selected, as opposed to PayPal Payments Standard, the store and PayPal are exchanging the $Token$ via redirections. Here, the pattern interrupted the chain of $Token$ when the user is redirected to PayPal for the payment. In both cases the oracle verified that the user U_2 has a confirmation and that the clause $paid(U_2, I') \wedge toStore(A)$ is satisfied. However, the oracle could not verify $\mathtt{O}(ack(U_2, I') \wedge onStore(A))$ because the events observed in the past were $\mathtt{O}(ack(U_2, I) \wedge onStore(A))$.

For these cases, the manual inspection confirmed that the order for the item I' was in the list of the orders in status “paid”. However, the balance sheet of the merchant shows that the payment for I' was done by U_1, the user used for the first user session, and not by U_2. In this case, U_1 authorized

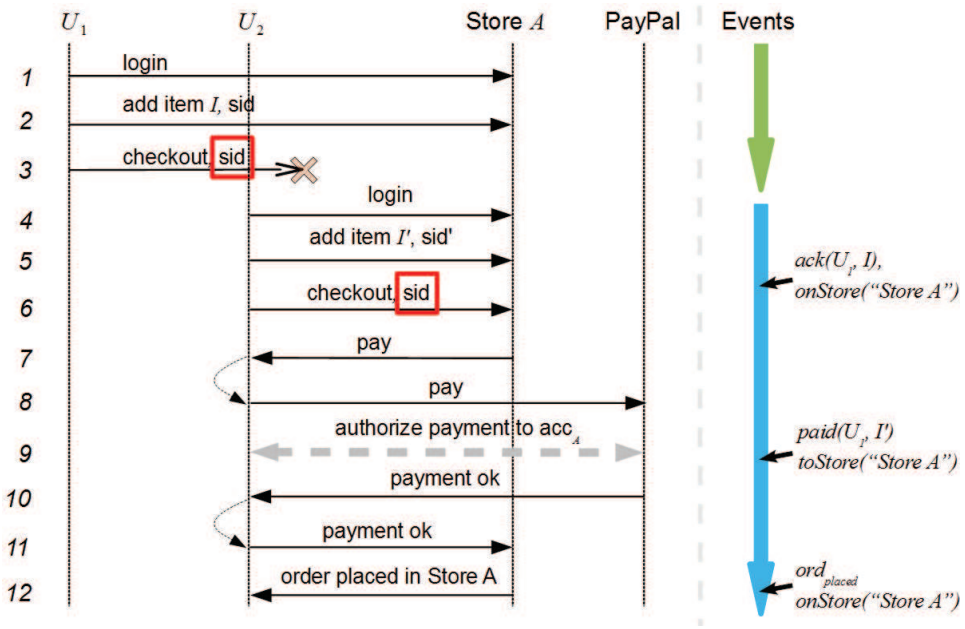


Figure 6.11: Session fixation in TomatoCart v.1.1.7

PayPal to pay for the amount of I while his/her credit card was charged for the amount of I'.

TomatoCart with PayPal Express Checkout - Session Fixation

Our prototype discovered a flaw in which a user could be authenticated as another user. The test cases were created by breaking the propagation chain of the parameter *sid* in two distinct points. Figure 6.11 shows one of them. The events of Figure 6.11 did not satisfy the logic formula because the payment I' was of a different amount than the one the user acknowledged I.

We investigated the problem and found out that *sid* carries the same value of the cookies and breaking it causes a *Session Fixation* in which, in our case, U_2 results logged in as U_1. From that point on U_2 can access U_1 data. As a consequence of the previous point, U_2 (now logged as U_1) pays the cart of U_1. However, we couldn't find any realistic exploitation of this vulnerability. Supposing that the victim (i.e. U_2) "clicks" on an URL crafted

by the attacker (U_1), then the victim could notice the fraud in three different moments (i) when checking the summary of order, (ii) when providing the shipping address (it shows the attacker's one), and (iii) during the payment because the amount is different.

6.9 Limitations

Our approach uses attack patterns that tamper with the observed dataflow and workflow. However, it does not test for other types of logic vulnerabilities such as unauthorized access to resources. Moreover, we did not consider cases in which the attacker can also play the role of a malicious store, or the cases in which the attacker can intercept and tamper with the messages between the application and the payment service. Nevertheless, the approach could be extended for detecting other types of logic vulnerabilities as well as supporting other types of attacks. This could be done by adding input traces of privileged user (e.g., admin), by adding other behavioral patterns, or by adding new attack patterns.

Second, the test generation favors efficiency over coverage. This means that only few values are used for each test category, to maximize the possibility to find bugs in a limited amount of time. A more thorough exploration of the attack space could be used to discover more vulnerabilities, however this could require a considerable amount of execution time. The focus of this chapter is to show how an automated approach can be used to find logic vulnerabilities in many real-word applications, and not to analyze in dept a single application (a scenario that would also require more input traces to better explore the application's logic).

Finally, we modeled logic properties in LTL. The use of LTL enables us to verify events with time dependency. However, LTL do not support algebra whose terms appears at different moment of the execution. For example, our oracles cannot verify whether the payment is the sum of the items the user added into the cart at some point in the past. There are works that extend LTL with constraints on integer numbers [BCF$^+$10], and they could be used by our oracle for checking more fine-grained properties.

6.10 Conclusions

In this chapter we showed that it is possible to automatically extract a simplified model of a web application, and that this model is sufficient to generate test cases that are likely to detect flaws in the logic of the web application.

The technique we presented aims at automating the manual testing in a single black-box tool. Our approach does not require the source code nor the specifications of the web application. It is based on two key concepts: model inference and an attack pattern-based test case generation. Our approach starting from a number of network traces infers a model, and then, it extracts a set of work flow and data flow patterns. Finally, it generates test cases following a number of attack patterns. The attack patterns reproduce the behavior of an attacker who intends to tamper with the data flow and the work flow of the application. We used our prototype to test seven eCommerce applications executing more than 3100 test cases, 900 of which violated the expected behavior of the application. As a result, our tool detected ten previously unknown logic vulnerabilities in the applications under test. Five of them allow an attacker to pay less or even shop for free.

Chapter 7
From Academia to Industry

This thesis was carried out within an industrial context. This allowed me to balance the design of novel security testing techniques with their pragmatic application to modern industrial-size scenarios. For example, we applied the methodology presented in Chapter 4 to support SAP engineers in the design and implementation of security protocols in SAP products. In this activity, we formally verified their design and implementation decision of the SAML SSO. Moreover, we developed a tool to automatize this type of analysis implementing the model checking-based security testing of Chapter 5. The tool has been used for the formal analysis and security testing of the SAP implementation of OAuth2.0 protocol.

Structure: This chapter is organized as follows. Section 7.1 introduces an excerpt of the formal analysis of the SAP implementation of SAML. Then, Section 7.2 presents the design verification and security testing tool implementing the techniques in Chapter 5 that was used for the SAP implementation of OAuth2 [Har12]. This tool was also used for the experiments in Chapter 5 and Chapter 4.

7.1 Formal Analysis of SAP NetWeaver New Generation Single Sign-On

Implementations of security protocols may deviate from the protocol specifications. However, deviations may endanger the overall security goals. For example, SAML-based SSO for Google Apps until 2008 neglected few but important message fields that allowed a malicious service provider to impersonate a user at any other service provider.

In Chapter 4, we showed that model checking can be used to analyze the security of authentication protocols by taking into account the different protocol flows and the different option configurations. In this section, we show how the technique of Chapter 4 can be further extended to include the design and implementation decisions. We applied this technique to the SAP implementation of SAML SSO supporting development units in taking decisions of the design.

7.1.1 SAP NetWeaver New Generation Single Sign-On

SAP NetWeaver New Generation Single Sign-On (hereafter NGSSO) is a component of the SAP software ecosystem implementing SAML SSO. The implementation provides software components to integrate services into the federated environment of a SAP customer.

NGSSO offers a configuration environment in which the administrator sets up SAML federated environments. The configuration of a SAML environment includes the list of identity and service providers, the supported profiles, as well as the entity configuration per single profile. For instance, the administrator could set up an identity provider that accepts signed messages in the SAML SSO. The configuration environment allows to establish the trust relationship between entities by setting up the certificate manager with entities' certificates. Moreover, the administrator can specify whether to use SSL/TLS or plain-text TCP sockets. The default values for all these configuration options reflect the recommendations and requirements of the SAML standard. However, administrators can change them according to the

requirements of the deployment landscape.

NGSSO implements the main SAML SSO flows and features the SAML SSO configuration options ranging from optional message fields, use of SSL/TLS at transport layer, and application of encryption and digital signature. We discussed these options in Chapter 4. In addition to these options, developers may consider additional features or deviations from the protocol specifications. In this thesis we describe two of them.

The first is a deviation from the SAML SSO specification in which the SP is stateless. SAML SSO prescribes that SPs must verify whether the ID of a response is equal to the ID they issue in the request. As a result, SPs keep an internal table where to store the authentication requests. We call this type of SP stateful SPs. However, stateful SPs may be vulnerable to Denial-of-Service attacks. For example, the attacker can request resources to the SP until the table is full or the SP runs out of memory. In security-sensitive scenarios, stateless SPs are preferred to stateful as more resistant to this type of denial of service.

The second feature is whether the SP uses session cookies during the protocol execution. SAML SSO does not use HTTP cookies at any step of the protocol. However, SP may use them to enforce particular policies. For example, SP would like to enforce that the client forwarding a SAML response is the same client that asked for the SAML request.

Our analysis included other features, however in this thesis we will not present all of them.

7.1.2 Analysis

Table 7.1 shows the options, the deviations, and the results of our analysis. Table 7.1 is structured as follows. Each row is a model with unique identifier *MID*. The column *from* is a pointer to the model from which *MID* is derived. The remaining columns are grouped in *SAML SSO options*, *Dec.*, and *Attacks* respectively for the options described in Chapter 4, the implementation decisions, and the result of the analysis. We use y when the option (resp. decision) is used or when the model checker found a violation; we use

		SAML SSO options										Dec.		Attacks	
		ARP		SSL/TLS			Sign								
MID	from	AReq	AResp	C-SP:AReq	ARP: AReq	ARP:AResp	AReq	AResp	ArtResolve	ArtResp	Encr. AA	SP sets cookie	SP checks ID	G1	G2
0fc	-	n	n	n	-	-	n	n	-	-	n	n	y	y	n
2fc	0fc	n	n	y	-	-	n	n	-	-	n	n	y	y	n
4fc	2fc	n	n	y	-	-	y	n	-	-	n	n	y	y	n
5fc	4fc	n	n	y	-	-	y	y	-	-	n	n	y	y	n
...															
0a-fc	0fc	n	n	n	-	-	n	n	-	-	n	y	y	n	n
5a-fc	5fc	n	n	y	-	-	y	y	-	-	n	y	n	n	n
...															

Table 7.1: Results for the SP-initiated profile

n otherwise.

We used the AVANTSSAR platform for the analysis with the HLPSL++ connector and SATMC validator. We wrote in total 85 formal specifications in HLPSL++ capturing the standard configuration options of Section 4.2.2, and the SAP internal design and implementation choices. Our analysis considered two execution scenarios. The first scenario involved only SAP participants, while the second considered an IdP by SAP and a standard SP. The security properties as well as the security assumptions are the same of Chapter 4.

Table 7.1 contains part of the results our analysis. It shows only the SP-initiated models and 2 out of 8 implementation decisions we considered. The first group of models, e.g. *0fc*, *2fc*, *4fc*, and *5fc* are the same in Chapter 4. The second group of models are new models and they are variation of the SAML SSO protocol. For example, the model *0a-fc* derives from the model *0fc* by adding the use of cookies.

Table 7.1 shows the following results. First, the protocol does not satisfy the property G1. Second, the standard protocol options are not sufficient

for fixing the flaw. Third, the use of cookies solves the vulnerability. Fourth, the two implementation decisions do not endanger the security with respect to the properties G1 and G2. Finally, the security goal G2 is always reached.

Developers can use the results of Table 7.1 for taking decisions about the design and the implementation. For example, in security-sensitive scenarios, they may enforce the use of cookie and avoid storing the ID as a Denial-of-Service countermeasure.

7.2 A Formal Analysis and Security Testing Tool

This section presents a tool that leverages on the design verification and security testing techniques of Chapter 4 and Chapter 5, and extends them to support developers in analyzing the security of security protocols. The tool helps developers, software engineers, and security experts in taking decisions during the development process and to detect flaws both at the design and deployment phases.

The current version of the tool targets web-based security protocol. However, we plan to extend it with the technique of Chapter 6. Our tool is a set of Eclipse plugins that supports (i) the specification of protocol options and implementation decisions, (ii) implements the design verification and model-based security testing, and (iii) supports the verification of multiple models and the execution of different tests.

Our tool was used for the experiments of Chapter 5 and it is currently used for supporting SAP developers and engineers in assessing the security of SAP implementation of OAuth2.

7.2.1 Design Verification

The design verification implements the formal analysis of security protocols via model checking. The process consists of three steps. First, the user writes the formal model and specify the security properties. Then, the model checker explores the model looking for violation of the property. If a violation is discovered, the model checker returns a counterexample witnessing the

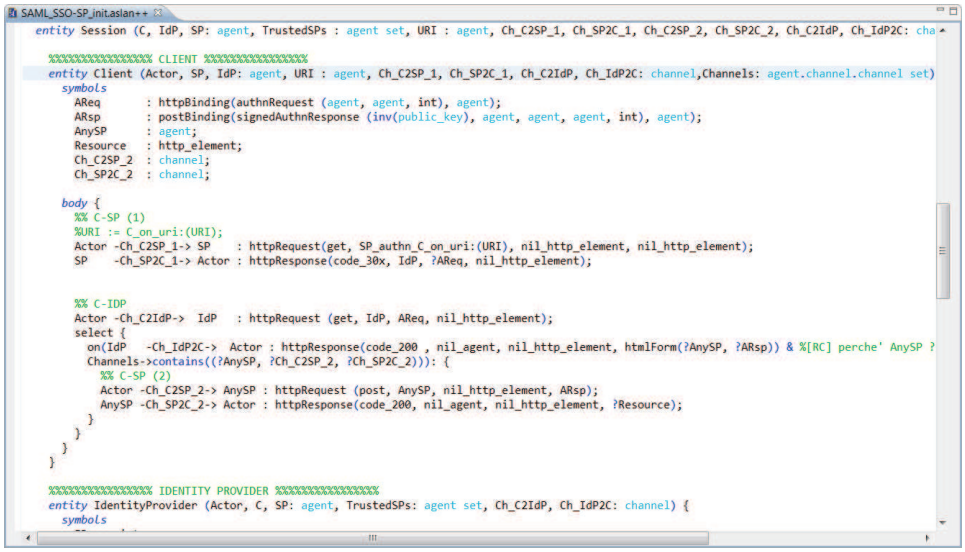


Figure 7.1: ASLan++ Editor

violation. Finally, the user inspects the counterexample and interprets it.

The tool supports these three steps by integrating parts of the AVANTSSAR platform. The AVANTSSAR platform offers as well a user interface for this workflow. However, it is available only as a web application or a set of command line tools. Our tool aims at integrating AVANTSSAR tools into a development environment familiar to engineers and developers.

Modeling

Our tool supports the ASLan and ASLan++ formal languages. However, it can be extended to support other languages such as HLPSL [CCC+04], Promela [Hol97], or ProVerif [Bla01]. The ASLan and ASLan++ editors implement features that are typical of an IDE, such as syntax highlighting and error highlighting. Figure 7.1 shows the ASLan++ editor and the syntax highlighting features.

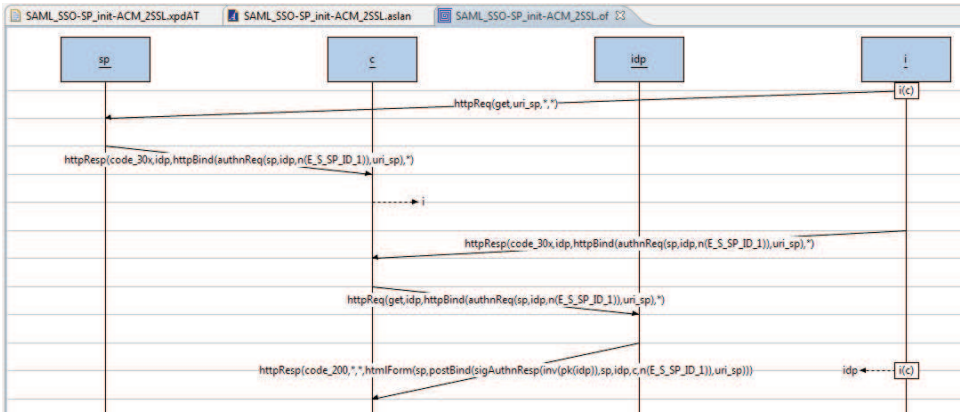


Figure 7.2: The Event Sequence Chart viewer

Verification

Our tool verifies formal models against the security properties via model checking. The tool currently integrates the SAT-based Model Checker (SATMC [ACC07]) that is executed within the Eclipse workspace. The integration is done via a programming interface that can be extended to support the other model checkers such as OFMC [BMV03] and ClAtSe [Tur06a], and SPIN [Hol97].

Visualization

If the model checker discovers a violation, it returns a counterexample. A counterexample is a sequence of messages that are sent and received by the protocol principals. Our tool offers a viewer called *Event Sequence Chart viewer* (ESC), to show a graphical representation of the counterexample similarly as seen in Chapter 4. Figure 7.2 shows the ESC viewer. The viewer displays a timeline for each of the principal, e.g., sp, c, idp, and the attacker i. Then, it places messages and arrows to show the direction of the communication. The viewer uses a dashed arrow when the message has been sent but not received yet.

The viewer offers other features that are not showed in Figure 7.2 such as attack trace inspection. In particular, it allows the user to inspect the original output returned by the model checker by simply clicking on the

messages displayed in the ESC.

7.2.2 Model-based Security Testing

The model-based security testing enables the user to generate and execute test cases for detecting vulnerabilities in real implementations. This process consists of five steps. First, the user models the protocol. Second, it uses specific algorithms for generating test cases. In the third step, the user defines the implementations under test. Finally, the test cases are executed and the results are shown to the user.

Modeling

The user develops the models as seen before for the design verification. Moreover, we plan to integrate the inference technique of Chapter 6.

Test Case Generation

The current version of the tools use SATMC for generating test cases as showed in Chapter 5. However, we plan to add the attack pattern-based test case generation algorithm of Chapter 6.

Implementation Under Test

The user specifies the implementations under test (IUTs) by using the IUT editor (see, e.g., Figure 7.3). The IUT is a data structure containing the mapping between model symbols and concrete values, and the set of protocol participants under test. Figure 7.3 shows the main parts of the UI in which the user inputs the mapping, and lists the participants under test.

The list of adapters to be used is included into the mapping table. For example, in Figure 7.3 we marked the Java classes mapping the abstract ASLan symbols. Our tool allows the user to implement customized adapters.

Adapter As said in Section 5.3, the ASLan symbol used for modeling protocol messages are associated to a set of program functions called constructor

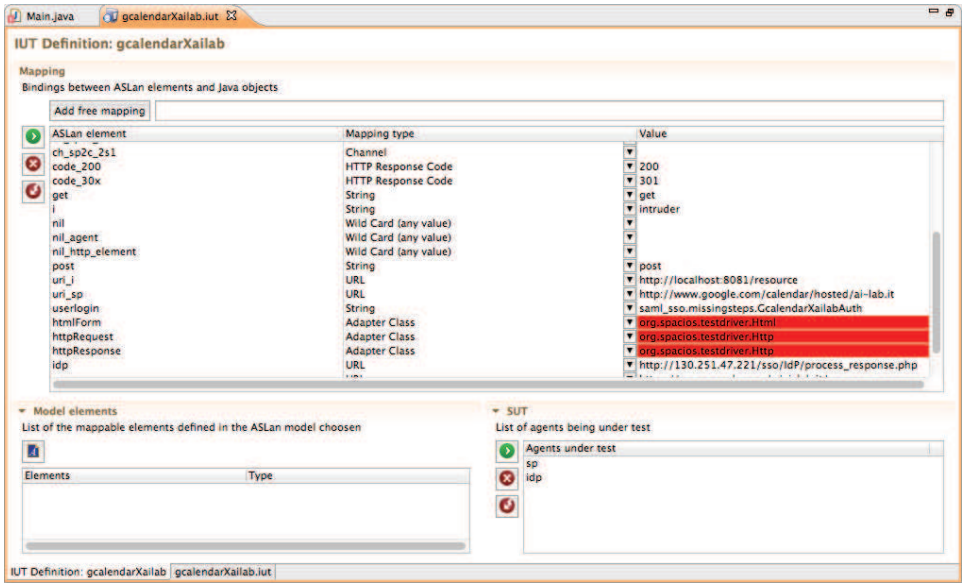


Figure 7.3: The IUT for testing the SAML-based SSO for Google Apps

and selectors. These functions are implemented into the adapter module of Figure 5.1.

Our tool enables user to develop custom adapters. An adapter is a Java class that comply with the following convention. First, the class is a public static Java class. Second, the constructor and parsers functions are static Java method of the same arity n of the ASLan symbol. Finally, the name of constructor methods is `constr_`sym and the name for parser is `pi`j`_`sym where sym is the name of the ASLan symbol and $0 \leq j \leq n$ is the position of the parameter. For example, let us consider the user-specified symbol `httpReq` in Section 4.1.2. The adapter for generating and parsing HTTP requests is the following:

```
1 public static Http {
2
3   // CONSTRUCTOR
4   public static HttpRequest constr_httpReq(
5     Object methodStr, Object hostPortStr,
6     Object urlParamStr, Object bodyUrlEncFormEnt) {
7     // [...]
8     }
```

```
9
10    // PARSERS
11    public static String pi1_httpReq(Object request){
12      // [...]
13      }
14
15    public static String pi2_httpReq(Object request){
16      // [...]
17      }
18
19    public static String pi3_httpReq(Object request){
20      // [...]
21      }
22
23
24    public static String pi4_httpReq(Object request){
25      // [...]
26      }
27  }
```

Test Case Execution

Our tool implements the instrumentation technique described in Chapter 5 for the concretization and execution of test cases. Given a model and a IUT, the instrumentation technique generates a set of Java program fragments encoding how to generate and parse protocol messages. A concrete test case is defined as an abstract counterexample and the set of corresponding program fragments.

Given a concrete test case and an IUT, the test execution engine interprets the counterexamples and executes the program fragments accordingly. Moreover, the test execution engine logs the HTTP messages exchanged with the protocol participants under test. The algorithm is shown in Section 6.5.

Visualization

Our tool allows the user to inspect the messages exchanged between the test execution engine and the implementation under test. In particular, it implements a view of the HTTP conversations and an HTTP messages viewer. The former offers a synthesis of the conversation listing the HTTP messages

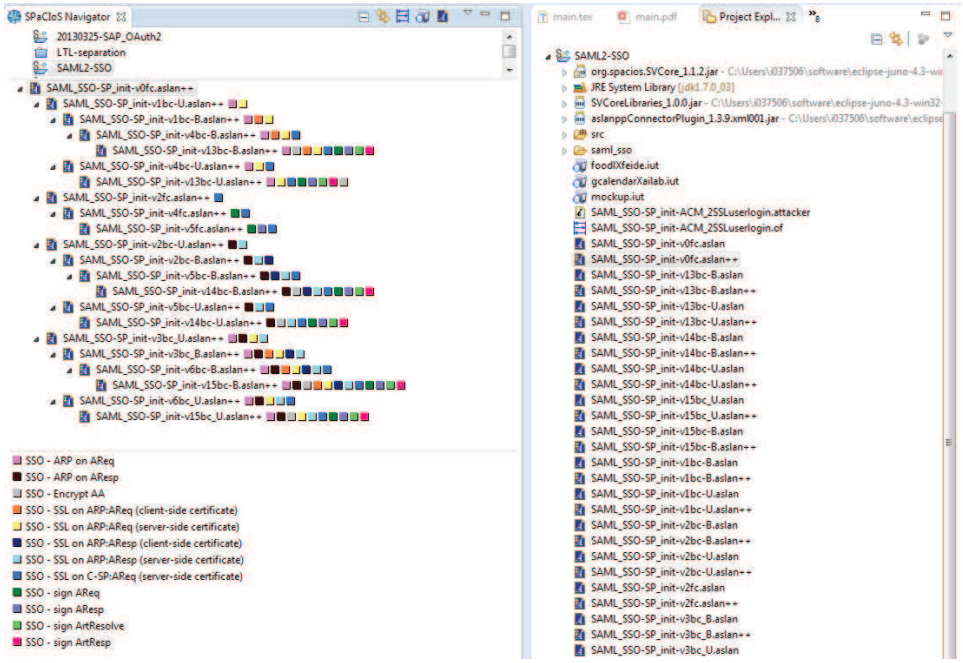


Figure 7.4: The Navigator

exchanged. Moreover, it links real HTTP messages with the abstract counterexample, allowing the user to deeply inspect the fields of the message. Our tool has a built-in web browser to visualize the content of HTTP responses.

7.2.3 Configuration and Implementation decisions

Our tool enables the specification of configuration options and implementation decisions. This is done through the *SPaCIoS navigator*. The navigator implements three main functionalities. First, it allows the specification of single protocol option (or decision) by means of *labels*. A label is a text description and a arbitrary color. Second, it allows for the creation of a new model (capturing the option) starting from an existing one. Finally, the navigator keeps track of all the model generated in a derivation tree in which the roots are the reference models. The tree and the labels are used later on for the preparation of the test/verification campaign.

The left-hand side of Figure 7.4 is the navigator. The upper part displays

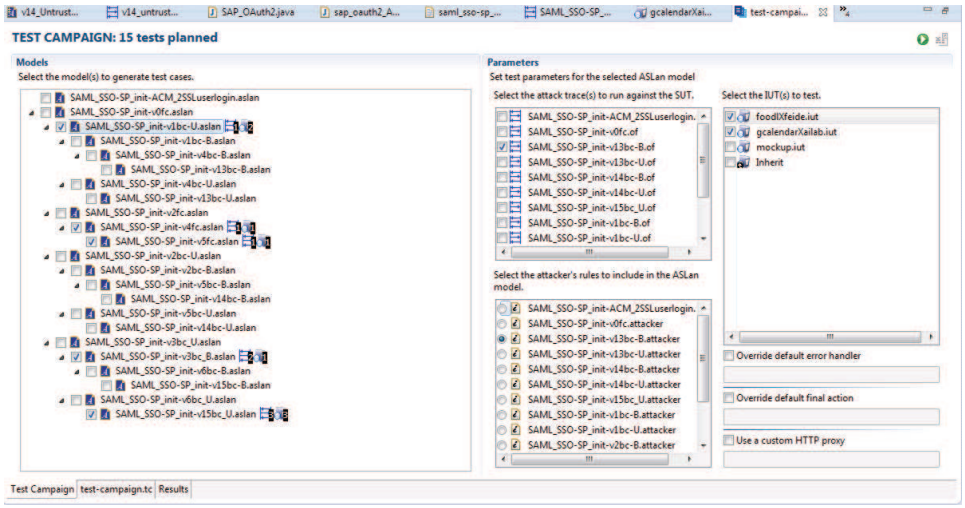


Figure 7.5: The Test Campaign Manager

the derivation tree in which each model (i.e., node tree) is associated to labels. A model can have more than one label. The lower part of the navigator shows the list of labels created during the analysis. They capture the configuration options of the SAML SSO standard. For comparison, the right-hand side of Figure 7.4 shows the standard view of Eclipse projects.

7.2.4 Verification and Test Campaign

A verification campaign is a multiple execution of the verification workflow. This solves the practical problem of verifying several models. Similarly, the test campaign consists of the executions of several test cases.

Figure 7.5 shows the editor for the test campaign manager. On the left-hand side, the editor displays the available models. Models are shown in a tree-like form. On the right-hand side, the editor shows the list of test cases generated and the IUTs available. The user selects the test cases and the IUTs, and she runs the campaign. At the end of the execution, the tool displays the HTTP conversations for off-line analysis. The result of a campaign is organized into tables. In addition, the tool logs the results and HTTP messages of all the test for future inspections.

The result of a verification and/or test campaign is organized into tables together with the result and/or test verdict as well as the labels (if any).

7.3 Conclusions

In this chapter we showed how some of the techniques presented in this dissertation have been transferred to SAP. We described the security analysis of the SAP implementation of SAML SSO, supporting developers in taking design and implementation decisions. In addition, we presented the design of a tool that eases the security analysis of protocol design, the assessment of protocol configuration, and the analysis of protocol deviation. In addition, the tool enables to test real implementations using counterexamples as abstract test cases.

Chapter 8
Conclusions and Future Work

In this chapter we summarize the contribution of this thesis with respect to the objectives that we have set in Section 1.1. Then, we give an overview of possible future work that could be carried out based on the results presented in this thesis.

8.1 Contributions

State-of-the-art security testing technologies do not provide automated support to the discovery of logic vulnerabilities in multi-party business applications. In this thesis, we have addressed the shortcomings of these technologies in order to support the automated detection of logic flaws.

We started in Chapter 4 with the design verification via model checking of the SAML SSO and OpenID authentication protocols. Starting from the specifications written in natural language, we wrote formal models capturing the behavior of the protocol participants, message structure, and composition of participants. We showed that when formal models are available, model checking can automatically discover flaws into the logic of the protocol design. However, the discoveries are not directly applicable to the real implementations. Moreover, we showed that there is still a substantial amount of manual work required to confirm the presence of the flaw in real implementations. Finally, we discovered that the design flaw can be exploited as a launching pad of XSS attacks in the SAML-base SSO for Google Apps.

All our findings have been discussed with members of the OASIS Security Services Technical Committee and a SAML V2.0 Errata has been redacted and approved [OAS12].

In Chapter 5 we tackled the first objective of this thesis that is testing real implementations starting from the attacks returned by a model checker. We proposed an approach that fills the gap between formal model and real implementations by the means of *model instrumentation*. The model instrumentation calculates a set of program fragments that encode the message generation, message parsing, and the check of the incoming messages against the current state of the participants. The fragments are then executed in the order established by the counterexample.

The approach of Chapter 4 and Chapter 5 is applicable when the specifications are available. In Chapter 6, we proposed an automated black-box approach that does not require a model as input. Our approach infers a model from a set of network traces. Afterwards, the model is used to generate test cases following a number of attack patterns. Finally, tests are executed against the real implementation and an oracle decides whether a property of the application has been violated.

This thesis has been carried out in an industrial context. This allowed us to balance the design of testing techniques with their pragmatical application to real world applications. The techniques of Chapter 4 and Chapter 5 have been implemented in an industrial tool, while the black-box testing technique of Chapter 6 is implemented as a proof-of-concept. The former tool has been used to test three implementations of SAML SSO and two of OpenID detecting the logic flaws discovered by the model checker. Moreover, this tool has been used to support SAP engineers to evaluate the security of the design of their SAML SSO implementation. Furthermore, it is currently used to test the SAP implementation of OAuth 2.0. The second tool implements the black-box testing approach described in Chapter 6. The tool has been used to test 12 eCommerce web application deployments discovering ten previously unknown critical vulnerabilities and about 900 presentations bugs. All the critical vulnerabilities that our techniques discovered have been responsibly disclosed.

8.2 Future Work

The results of this thesis corroborate the claim that model-based testing can improve the effectiveness of existing security testing methodologies. As a future work, we would suggest to strengthen the basis of the claim in two ways. First, the techniques of this thesis could be extended to detect newer vulnerability classes that are still discovered by manual inspection, e.g., improper authentication and authorization, and session management vulnerabilities. Second, it would be interesting to explore how our approach could be applied to other types of business applications, e.g., billing and invoicing applications.

In this thesis, we argued that the capability of detecting vulnerabilities relies on two factors. First, the ability to generate good tests and second, the ability to decide whether the test execution proves the presence of a vulnerability. This dissertation mainly focused on the former, while the latter is implemented by the means of satisfiability of a user-provided LTL formula modeling the expected behavior. However, the development of LTL formula can be error-prone. In addition, the development of the LTL formula may require application-specific knowledge for the extraction of symbols from the test executions. As a future direction, it would be interesting to investigate on the automatic generation of the expected behavior. For example, this could be achieved by using Daikon [EPG+07] to generate likely invariants or by inferring LTL formulas from the dataflow and workflow behavioral patterns.

In this thesis, we showed the use of model checking to detect logic flaws in protocol implementations. As opposed to model checking, the attack pattern-based approach uses heuristics instead of exhaustive search. The experimental results in Chapter 6 indicate that this technique is efficient. However, the empirical evidence should be supported with theoretical arguments. In particular, it would be interesting to compare quantitatively the two test generation approaches.

Furthermore, it would be interesting to apply the attack pattern-based approach to generate test cases to test security protocol implementations.

This may require to extend the set of attack patterns targeting security protocol attacks and session management vulnerabilities. Similarly, it would be interesting to extend the application of model checking techniques to the black-box testing scenario to detect violations in eCommerce applications. This could be done by translating the navigation graph and the behavioral pattern in a formal language (e.g., ASLan or ASLan++) and then by using a model checker in the classical sense in order to generate test cases.

References

[AAA+12] Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Gabriel Erzse, Simone Frau, Marius Minea, Sebastian Mödersheim, David von Oheimb, Giancarlo Pellegrino, Serena Elisa Ponta, Marco Rocchetto, Michaël Rusinowitch, Mohammad Torabi Dashti, Mathieu Turuani, and Luca Viganò. The avantssar platform for the automated validation of trust and security of service-oriented architectures. In Cormac Flanagan and Barbara König, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2012.

[ABB+05] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P. C. Heám, O. Kouchnarenko, 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 *Proceedings of the 17th international conference on Computer Aided Verification*, CAV'05, pages 281–285, Berlin, Heidelberg, 2005. Springer-Verlag.

[AC02] Alessandro Armando and Luca Compagna. Automatic sat-compilation of protocol insecurity problems via reduction to planning. In DoronA. Peled and MosheY. Vardi, editors, *Formal Techniques for Networked and Distributed Sytems - FORTE 2002*, volume 2529 of *Lecture Notes in Computer Science*, pages 210–225. Springer Berlin Heidelberg, 2002.

[ACC07] A. Armando, R. Carbone, and L. Compagna. Ltl model checking for security protocols. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, pages 385–396, July 2007.

[ACC+08] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, and Llanos Tobarra Abad. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In *Proceedings of ACM FMSE08*, 2008.

[ACPP11] Wihem Arsac, Luca Compagna, Giancarlo Pellegrino, and Serena Elisa Ponta. Security validation of business processes via model-checking. In Úlfar Erlingsson, Roel Wieringa, and Nicola Zannone, editors, *ESSoS*, volume 6542 of *Lecture Notes in Computer Science*, pages 29–42. Springer, 2011.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2), November 1987.

[AP09] Alessandro Armando and Serena Elisa Ponta. Model checking of security-sensitive business processes. In Pierpaolo Degano and Joshua D. Guttman, editors, *Formal Aspects in Security and Trust*, volume 5983 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2009.

[ASW98] N. Asokan, V. Shoup, and M. Waidner. Asynchronous protocols for optimistic fair exchange. In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 86–99, 1998.

[BBGM10] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010.

[BCF+10] Marcello M. Bersani, Luca Cavallaro, Achille Frigeri, Matteo Pradella, and Matteo Rossi. Smt-based verification of ltl specifications with integer constraints and its application to runtime checking of service substitutability. *CoRR*, abs/1004.2873, 2010.

[BCFV07] Davide Balzarotti, Marco Cova, Viktoria V. Felmetsger, and Giovanni Vigna. Multi-module vulnerability analysis of web-based applications. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, New York, NY, USA, 2007. ACM.

[BFHS03] Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation specification: a new approach to design and analysis of e-service composition. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, pages 403–410, New York, NY, USA, 2003. ACM.

[BG05] Michael Backes and Thomas Gross. Tailoring the dolev-yao abstraction to web service realities. In *ACM SWS05*, 2005.

[Big13] BigCommerce Pty. Ltd. BigCommerce Interspire, 2013.

[BJR08] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines using domains with equality tests. In *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, FASE'08/ETAPS'08, Berlin, Heidelberg, 2008. Springer-Verlag.

[Bla01] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of CSFW'01*, pages 82–96. IEEE CSP, 2001.

[BMPV06] Michael Backes, Sebastian Mödersheim, Birgit Pfitzmann, and Luca Viganò. Symbolic and Cryptographic Analysis of the Secure WS-ReliableMessaging Scenario. In *Proceedings of FOSSACS'06*, 2006.

[BMV03] D. Basin, S. Mödersheim, and L. Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. Submitted, available at `www.infsec.ethz.ch/publications/ofmc.pdf`, 2003.

[BOP11] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. Security mutants for property-based testing. In *TAP 2011*, 2011.

[BOP12a] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. Semi-automatic security testing of web applications from a secure model. In *SERE*. IEEE, 2012.

[BOP12b] Matthias Buchler, Johan Oudinet, and Alexander Pretschner. Spacite – web application testing engine. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:858–859, 2012.

[CCC+04] Yannick Chevalier, Luca Compagna, Jorge Cuellar, Paul Hankes Drielsma, Jacopo Mantovani, Sebastian Mödersheim, and Laurent Vigneron. *A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols*, volume 180 of *Automated Software Engineering*, pages 193–205. Austrian CS, Austria, September 2004.

[CCGR99] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Verifier. *LNCS 1633*, 1999.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[CJ97] John Clark and Jeremy Jacob. A Survey of Authentication Protocol Literature: Version 1.0. `www.cs.york.ac.uk/~jac/papers/drareview.ps.gz`, November 1997.

[CLS00] Rance Cleaveland, Tan Li, and Steve Sims. The concurrency workbench of the new century, version 1.2 - user's manual, 2000.

[Cro06] Douglas Crockford. RFC4627: The application/json Media Type for JavaScript Object Notation (JSON), July 2006.

[DBKV11] Adam Doupé, Bryce Boe, Christopher Kruegel, and Giovanni Vigna. Fear the ear: discovering and mitigating execution after redirect vulnerabilities. In

Proceedings of the 18th ACM conference on Computer and communications security, CCS '11, New York, NY, USA, 2011. ACM.

[DCKV12] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the State: A State-Aware Black-Box Vulnerability Scanner. In *Proceedings of the 2012 USENIX Security Symposium (USENIX 2012)*, Bellevue, WA, August 2012.

[DCV+05] Davis D, Ferris C, Gajjala V, Gavrylyuk K, Gudgin M, Kaler C, Langworthy D, Moroney M, Nadalin A, Roots J, Storey T, Vishwanath T, , and Walter D. Secure ws-reliablemessaging scenarios, April 2005.

[DCV10] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In Christian Kreibich and Marko Jahnke, editors, *DIMVA*, volume 6201 of *LNCS*. Springer, 2010.

[DHK11] F. Dadeau, P.-C. Héandam, and R. Kheddam. Mutation-based test generation from security protocols in HLPSL. In *ICST 2011*, 2011.

[DHL01] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. Business process coordination: State of the art, trends, and open issues. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 3–13, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[DHP09] Arnaud Dury, Hesham H. Hallal, and Alexandre Petrenko. Inferring behavioural models from traces of business applications. In *Proceedings of the 2009 IEEE International Conference on Web Services*, ICWS '09, Washington, DC, USA, 2009. IEEE Computer Society.

[DNL99] Ben Donovan, Paul Norris, and Gavin Lowe. Analyzing a library of security protocols using Casper and FDR. In *Proceedings of the FLOC'99 Workshop on Formal Methods and Security Protocols (FMSP'99)*, 1999.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[Fac13] Facebook Inc. Facebook Connect, 2013.

[FBS04] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting bpel web services. In *Proceedings of WWW '04*, 2004.

[FCKV10] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, Berkeley, CA, USA, 2010. USENIX Association.

[FDR97] FDR2 — Failures-Divergence Refinement, Documentation. `http://www.fsel.com/documentation/fdr2/html/index.html`, 1997.

[FF13] Fyodor and David Fifield. SecTools.Org: Top 125 Network Security Tools, 2013.

[FHBH+99] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999.

[FKK11] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), August 2011.

[FWA09] Gordon Fraser, Franz Wotawa, and Paul Ammann. Issues in using model checkers for test case generation. *J. Syst. Softw.*, 82(9):1403–1418, September 2009.

[GGJ+13] Bai Guangdong, Meng Guozhu, Lei Jike, Sathyanarayan Venkatraman Sai, Saxena Prateek, Sun Jun, Liu Yang, and Dong Jinsong. Authscan: Automatic extraction of web authentication protocols from implementations. In *Annual Network & Distributed System Security Symposium (NDSS), 2013.* The Internet Society, 2013.

[Goo08] Google. SAML Single Sign-On (SSO) Service for Google Apps, 2008.

[Goo09] Google. Google security and product safety, 2009. [Online; accessed 16-July-2012].

[Hac13] Hackers for Charithy. The Google Hacking Database at Hacking for Charity, 2013.

[Har12] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), October 2012.

[HGOR13] K. Hossen, R. Groz, C. Oriat, and J.-L. Richier. Automatic generation of test drivers for model inference of web applications. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 441–444, 2013.

[HGR11] Karim Hossen, Roland Groz, and Jean-Luc Richier. Security vulnerabilities detection using model inference for applications and security protocols. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, march 2011.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Hol97] G. J. Holzmann. The Spin Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[Hol04] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[Hor51] Alfred Horn. On sentences which are true of direct unions of algebras. *J. Symb. Log.*, 16(1):14–21, 1951.

[HVO06] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, March 2006.

[HYH+04a] Yao-Wen Huang, Fang Yu, C. Hang, Chung-Hung Tsai, D.T. Lee, and Sy-Yen Kuo. Verifying web applications using bounded model checking. In *Dependable Systems and Networks, 2004 International Conference on*, pages 199–208, 2004.

[HYH+04b] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[JKK06] Nenad Jovanovic, Christopher Krügel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, pages 258–263. IEEE Computer Society, 2006.

[JKK10] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *J. Comput. Secur.*, 18(5):861–907, September 2010.

[JRV00] Florent Jacquemard, Michaël Rusinowitch, and Laurent Vigneron. Compiling and verifying security protocols. In *Proceedings of the 7th international*

conference on Logic for programming and automated reasoning, LPAR'00, pages 131–160, Berlin, Heidelberg, 2000. Springer-Verlag.

[Ker00] Kerberos: The Network Authentication Protocol, 2000. URL: `http://web.mit.edu/kerberos/www/`.

[KMR05] Lea Kutvonen, Janne Metso, and Toni Ruokolainen. Inter-enterprise collaboration management in dynamic business networks. In *Proceedings of the 2005 Confederated international conference on On the Move to Meaningful Internet Systems - Volume Part I*, OTM'05, pages 593–611, Berlin, Heidelberg, 2005. Springer-Verlag.

[KS11] Chulyun Kim and Kyuseok Shim. Text: Automatic template extraction from heterogeneous web pages. *Knowledge and Data Engineering, IEEE Transactions on*, 23(4), april 2011.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, Berkeley, CA, USA, 2005. USENIX Association.

[Low96] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Proceedings of TACAS'96*, 1996.

[Low97] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of of the 10th IEEE CSFW '97*. IEEE Computer Society Press, 1997.

[LX11] Xiaowei Li and Yuan Xue. Block: a black-box approach for detection of state violation attacks towards web applications. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, New York, NY, USA, 2011. ACM.

[MCJ97] W. Marrero, E. M. Clarke, and S. Jha. Model checking for security protocols. tech. report cmu-scs-97-139. Technical report, CMU, May 1997.

[Mic13] Microsoft Corporation. Microsoft Live ID Web Authentication SDK, 2013.

[MIT] MITRE. Common Weakness Enumeration.

[MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 365–383, New York, NY, USA, 2005. ACM.

[MMS97] J. C. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murphi. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 141–153, 1997.

[MSS98] J.C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of ssl 3.0. In *Seventh USENIX Security Symposium*, pages 201–216, 1998.

[MV09] Sebastian Mödersheim and Luca Viganò. The open-source fixed-point model checker for symbolic analysis of security protocols. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *FOSAD*, volume 5705 of *Lecture Notes in Computer Science*, pages 166–194. Springer, 2009.

[NAS05] NASA. Java PathFinder, 2005.

[nop13] nopCommerce. NopCommerce, 2013.

[Nov11] Novell. Access Gateway Appliance security concerns poisoning or tampering cookies, 2011. [Online; accessed 16-July-2012].

[NS78] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), December 1978.

[OAS08] OASIS Consortium. Security Assertion Markup Language V2.0 Tech. Overview. `http://wiki.oasis-open.org/security/Saml2TechOverview`, March 2008.

[OAS12] OASIS. SAML Version 2.0 Errata 05. `http://docs.oasis-open.org/security/saml/v2.0/sstc-saml-approved-errata-2.0.html`, May 2012.

[Ope07] OpenID Foundation. OpenID Specifications. `http://openid.net/developers/specs/`, 2007.

[Pay12a] PayPal. PayPal Express Checkout Integration Guide, August 2012.

[Pay12b] PayPal. PayPal Payments Standard Integration Guide, June 2012.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*. IEEE Computer Society, 1977.

[Pon12] Ponemon Institute. 2012 Cost of Cyber Crime Study. Technical report, Ponemon Institute, 2012.

[PST02] M. Panti, L. Spalazzi, and S. Tacconi. Using the nusmv model checker to verify the kerberos protocol, 2002.

[PVY01] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *J. Autom. Lang. Comb.*, 7(2):225–246, November 2001.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[Qui93] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Int., 1997.

[SBK12] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Have things changed now? an empirical study on input validation vulnerabilities in web applications. *Computers & Security*, 31(3):344–356, 2012.

[SBS04] G. Salaun, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. In *Web Services, 2004. Proceedings. IEEE International Conference on*, pages 43–50, 2004.

[SLS06] Andreas Schaad, Volkmar Lotz, and Karsten Sohr. A model-checking approach to analysing organisational controls in a loan origination process. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, SACMAT '06, pages 139–149, New York, NY, USA, 2006. ACM.

[SM99] Vitaly Shmatikov and John C. Mitchell. Analysis of a fair exchange protocol. In *Proceedings of the 1999 FLoC Workshop on Formal Methods and Security Protocols*, Trento, Italy, 1999.

[SN12] Giuseppe Scrivano and Hrvoje Niksic. GNU wget, 2012.

[spo02] Security Protocols Open Repository, 2002. `http://www.lsv.ens-cachan.fr/spore/index.html`.

[The04] The OWASP Foundation. OWASP top 10 application security risks - 2004, 2004.

[The07a] The Open Web Aapplication Security Project. The WebGoat Project, 2007.

[The07b] The OWASP Foundation. OWASP top 10 application security risks - 2007, 2007.

[The08] The OWASP Foundation. OWASP testing guide, January 2008.

[The10] The OWASP Foundation. OWASP top 10 application security risks - 2010, 2010.

[the12] the Skipfish Project. skipfish: Web Application Security Scanner, 2012.

[The13a] The jQuery Foundation. jquery, January 2013.

[The13b] The Mozilla Foundation. Mozilla Persona, 2013.

[The13c] The OWASP Foundation. OWASP top 10 application security risks - 2013, 2013.

[Tru13] Trustwave. 2013 Trustwave Global Security Report. Technical report, Trustwave, 2013.

[Tur06a] M. Turuani. The CL-Atse Protocol Analyser. In F. Pfenning, editor, *Proceedings of 17th Int. Conf. on Rewriting Techniques and Applications, RTA*, LNCS, Seattle (WA), August 2006. Springer.

[Tur06b] Mathieu Turuani. The cl-atse protocol analyser. In *Proceedings of the 17th international conference on Term Rewriting and Applications*, RTA'06, pages 277–286, Berlin, Heidelberg, 2006. Springer-Verlag.

[UNI10] UNINETT. simplesamlphp-1.6.3 is available, with a security fix, 2010. [Online; accessed 16-July-2012].

[Ver13] Verizon. 2013 Data Breach Investigations Report. Technical report, Verizon, 2013.

[vOM11] David von Oheimb and Sebastian Mödersheim. Aslan++ – a formal security specification language for distributed systems. In *Proceedings of the 9th international conference on Formal Methods for Components and Objects*, FMCO'10, pages 1–22, Berlin, Heidelberg, 2011. Springer-Verlag.

[w3a13] w3af. w3af: Web Application Attack and Audit Framework, 2013.

[WCW12] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing me onto your accounts through facebook and google: a traffic-guided security study of commercially deployed single-sign-on web services. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2012.

[WCWQ11] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to shop for free online – security analysis of cashier-as-a-service based web stores. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, Washington, DC, USA, 2011. IEEE Computer Society.

[Whi13] WhiteHat. Website Security Statistics Report. Technical report, WhiteHat, May 2013.

[WL93] Thomas Y. C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy*, SP '93, pages 178–, Washington, DC, USA, 1993. IEEE Computer Society.

[WMM09] Christian Wolter, Philip Miseldine, and Christoph Meinel. Verification of business process entailment constraints using spin. In *Proceedings of the 1st International Symposium on Engineering Secure Software and Systems*, ESSoS '09, pages 1–15, Berlin, Heidelberg, 2009. Springer-Verlag.

[Wor05] World Wide Web Consortium. Document object model, January 2005.

[Wor07] World Wide Web Consortium. Simple object access protocol (soap) 1.2, April 2007.

[WS07] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 32–41. ACM, 2007.

[XA06] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[XB05] Lai Xu and Sjaak Brinkkemper. Modeling multi-party web-based business collaborations. In Robert Meersman, Zahir Tari, Pilar Herrero, Gonzalo Méndez, Lawrence Cavedon, David Martin, Annika Hinze, George Buchanan, María S. Pérez, Víctor Robles, Jan Humble, Antonia Albani, Jan L. G. Dietz, Hervé Panetto, Monica Scannapieco, Terry A. Halpin, Peter Spyns, Johannes Maria Zaha, Esteban Zimányi, Emmanuel Stefanakis, Tharam S. Dillon, Ling Feng, Mustafa Jarrar, Jos Lehmann, Aldo de Moor, Erik Duval, and Lora Aroyo, editors, *OTM Workshops*, volume 3762 of *Lecture Notes in Computer Science*, pages 866–875. Springer, 2005.

[Xu07] Lai Xu. Outsourcing and multi-party business collaborations modeling. *JECO*, 5(2):77–96, 2007.

Appendices

Appendix A
Résumé en Français

Résumé

Le logiciel d'entreprise multi-partis sont des logiciels distribués sur le Web qui mettant en œuvre des fonctions collaboratives d'entreprise. Ces types de logiciels sont les principaux objectifs des attaquants qui exploitent les vulnérabilités de logiciels pour les activités malveillantes. La principale classe de vulnérabilités logicielles sont la conséquence de insuffisante validation d'entrée fournie par les utilisateurs. Récemment, un type moins connu de la vulnérabilité, les anomalies logiques, ont attiré l'attention des chercheurs. Sur la base de la disponibilité des documents, peut être utilisé deux techniques de testing: le model checking, les tests de sécurité de type "boîte noire". Malheureusement, le model checking ne prend pas en charge le test des implémentations actuelles, tandis que de tests de type boîte noire n'est pas assez sophistiquée pour découvrir les vulnérabilités logique. Dans cette thèse, nous présentons deux techniques d'analyse modernes visant à résoudre les inconvénients de état de l'art. Pour commencer, nous présentons la vérification de deux protocoles de sécurité modernes utilisant le model checking. Ensuite, nous nous concentrons sur l'extension du model checking pour soutenir les tests automatisés d'implémentations. La seconde technique consiste en un test de sécurité de boîte noire qui combine l'inférence du modèle, l'extraction du workflow et des data flow, et, à la fin, une technique de generatione des tests basés sur les modèles d'attaque. En conclusion, nous discutons l'application dans un contexte industriel des techniques développées dans cette thèse.

A.1 Introduction

Le logiciel d'entreprise multi-partis sont des programmes informatiques qui sont utilisés pour effectuer des fonctions commerciales. Aujourd'hui, les logiciels d'entreprise sont développées comme une composition de services de réseau. Chaque service implémente une fonction élémentaire qui est mis à disposition sur un réseau d'ordinateurs. Ces logiciels sont utilisés par browesers web, ou par les applications clientes qui s'exécutent sur des ordinateurs personnels, ou sur des appareils mobiles. A l'origine, les applications de l'entreprise étaient accessibles via des réseaux privés, mais sont aujourd'hui accessibles à travers les réseaux publics tels que, par exemple, l'Internet.

A.1.1 Security Risks of Multi-party Business Applications

La logiciels d'entreprise jouent un rôle important dans de nombreux domaines, et sont actuellement utilisés par des millions d'utilisateurs et organisations pour acheter des biens et services, effectuer des transactions monétaires, et de stocker des données confidentielles. Pour cette raison, les logiciels d'entreprise sont un objectif principale pour les attaquants cyber qui ont un intérêt à faire un large éventail d'activités criminelles.

A.1.2 The Rise of Logic Flaws

Les vulnérabilités les plus courantes sont causées par une validation insuffisante des entrées utilisateur, comme l'injection SQL (SQLI) et Cross-Site Scripting (XSS). Ces vulnérabilités logicielles ont été largement étudiés par la communauté scientifique. Un autre type de vulnérabilité qui sont mal étudié a récemment attiré l'attention des chercheurs. Ce type est provoqué par des erreurs logiques dans le logiciel.

La tendance générale d'incidents cyber qui sont causés par des erreurs logique, a augmenté ces dernières années. Le nombre de ces incidents en 2012 s'élève à 267, dont 82 dans les applications web de soleil. Le pic a été en 2008 avec 384 cas, dont 143 cas dans les applications web.

A.1.3 Objectives and Challenges

Les chercheurs ont proposé plusieurs techniques pour découvrir des vulnérabilités dans les logiciels d'entreprise multi-partis. Le choix de la technique dépend de l'information qui est disponible pour l'analyste. Cette information peut être le code source ou des modèles qui décrivent le comportement du logiciel. Cependant, les développeurs ne distribuent pas le code source à des sociétés tierces. Par conséquent, les techniques basées sur la disponibilité du code source ne peuvent pas être utilisés dans ce domaine particulier.

Lorsque les spécifications du logiciel sont disponibles, l'analyste peut utiliser la technique de vérification de modèle pour explorer les états d'un modèle formel et découvrir erreurs logiques. Cependant, la technique de vérification de modèle fournit pas de support pour la présence d'erreurs dans le logiciel. Enfin, à la fois comme le code source et les logiciels spécifications ne sont pas disponibles, vous pouvez utiliser les techniques de vérification de «boîte noire». Cependant, ces techniques n'ont pas la sophistication nécessaire pour trouver des erreurs logiques. Cette thèse vise à trouver une solution aux limitations ci-dessus. Plus précisément, les objectifs de cette thèse sont les suivants :

Objectif 1 :

> *Lorsque les modèles de logiciels sont disponibles, nous pouvons vérifier automatiquement si un logiciel souffre d'une faille logique qui a été découvert en utilisant la technique de la vérification de modèle ?*

Pour répondre à cette question, nous nous attendons à des défis. Le principal défi réside dans la traduction entre les éléments abstraits dans un modèle avec les éléments concrets du monde physique. La complexité de la traduction dépend de plusieurs facteurs, par exemple par le choix de l'interface pour interaggire avec le logiciel, à partir de la relation entre le modèle et le logiciel, et le type de vulnérabilités qui sont découverts.

Une interface au niveau trop élevé du logiciel peut enlever trop de détails dell'implementatione. Par conséquent, la traduction nécessite des algorithmes plus puissants pour reconstituer l'information manquante. D'autre part, une interface de bas niveau peut plus transmettre trop d'informations et en conséquence le modèle peut être trop détaillée, rendant les techniques modèle de contrôle inefficace.

La relation entre les modèles et les logiciels peuvent ne pas être nécessairement un-à-un. Par exemple, les spécifications d'un protocole standard de sécurité sont un modèle informel qui décrivent un nombre indéterminé de mises en œuvre qui ne sont pas nécessairement identiques. Dans la réalisation des essais, une traduction stricte empêcherait sa réutilisation sur d'autres logiciels en augmentant l'effort pour réaliser les tests.

En fin de compte, les règles de l'abstraction doivent prendre en considération le type de vulnérabilité que vous souhaitez découvrir. Par exemple, pour trouver des vulnérabilités XSS, l'analyste insère les paramètres d'entrée de malveillance dans une URL. Par la suite, l'analyste vérifie si la réponse du logiciel contient la même entrée. Si les règles de l'abstraction ignorent cette information dans la réponse, l'analyste peut ne pas être en mesure de juger si le logiciel est vulnérable.

Objectif 2 :

> *Lorsque les modèles ne sont pas disponibles, il est possible de découvrir des vulnérabilités dans la logique du logiciel automatiquement*

Pour découvrir des vulnérabilités dans la logique de logiciels, nous avons besoin de deux types de modèle. Le premier type de modèle et d'un modèle de comportement du logiciel. Le deuxième modèle est une description de la logique qui met en oeuvre le logiciel. Ce modèle peut être obtenu par le modèle inférence algorithmes déduire un tel modèle en utilisant les observations sur le logiciel.

La deuxième difficulté est lié à la performance des techniques de techniques de test avec le modèle. Pour découvrir les vulnérabilités logique, nous

avons besoin des algorithmes qui sont conscients des états internes du logiciel et de la logique mise en œuvre. Ces algorithmes peuvent être automatisées techniques de raisonnement, comme la technique de vérification de modèle. Cependant, les techniques basées sur des modèles comme le model checking souffrent du problème connu comme l'explosion d'états, dans lequel l'espace d'état à explorer pourrait etre assez grande pour rendre la tache impossible.

A.2 Case Studies

Cette thèse utilise deux études de cas pour présenter deux nouvelles techniques de test automatique. La première classe concerne les protocoles d'authentification unique. Deux protocoles, OASIS Security Assertion Markup Language 2.0 Web browser Single Sign-On (SAML SSO) et OpenID Authentication Protocol (OpenID) serviront d'exemples pour illustrer la détection de failles lorsqu'une spécification de l'application est disponible publiquement. La seconde classe concerne les applications de commerce en ligne. Ces applications seront utilisées pour illustrer les techniques de détection de failles selon une approche "boite noire", envisageable lorsque les spécifications des applications à analyser ne sont pas disponibles.

A.2.1 Case Study 1 : Web-based Single Sign-On Protocols

SSO SAML et OpenID sont deux protocoles de sécurité qui permettent aux partenaires commerciaux d'identifier leurs utilisateurs à la fois, puis de leur permettre d'accéder aux services du logiciel de l'entreprise sans avoir besoin de les identifier à nouveau. Les implémentations de SAML SSO et OpenID font partie de la célèbre des logiciels tels que SAP NetWeaver Identity Manager, IBM Tivoli Federated Identity Manager, et Google Apps (Gmail et Google Calendar). Chaque jour, des millions d'utilisateurs sont identifiés à l'aide de ces deux protocoles. Par exemple, Google affirme que plus de 5

millions organisations utilisent OpenID et SAML SSO pour identifier leurs employés.

OpenID et SAML SSO fournissent trois rôles : un client C, un IdP de fournisseur d'identité, et un SP de fournisseur de services. Le but de C, typiquement un navigateur web entraînée par un utilisateur, est d'accéder à un service ou une ressource offerte par SP. IdP authentifie C et de créer une affirmation d'authentification (un message spécial utilisé pour identifier les utilisateurs). Les protocoles d'identification finissent quand SP consomme l'affirmation généré par l'IdP, et fournit C la ressource demandée.

Á travers une comparaison entre l'identification unique avec les schémas classiques basées sur de multiples mots de passe, il devient naturel de s'attendre à ce que le SSO SAML et OpenID offrent une propriété de l'authentification mutuelle entre C et SP.

A.2.2 Case Study 2 : eCommerce Applications

Applications Web de commerce électronique sont des produits logiciels conçus pour vendre et acheter des biens et des services sur le Web. Applications Web mettent en œuvre des catalogues virtuels et des caddies virtuels grace à laquelle les clients choisissent les produits qu'ils envisagent d'acheter. Ces logiciels ont un front-end pour les clients et un back-end pour les employés et les administrateurs. Les clients de créer des commandes via le front-end. Les employés utilisent le back-end pour traiter les commandes, collecter les marchandises du magazin, et préparer l'expédition.

Les applications Web à intégrer les systèmes de paiement de commerce électronique offerts par des tiers. Ceci est mis en pratique par le biais des interfaces de programmation (API) offerts par des services tels que PayPal, Amazon Payments, Google Checkout, ou Authorize.NET. L'intégration des services peut être effectuée à différents points dans le processus d'achat et dépend également du type de système de paiement choisi.

A.3 Model Checking

Lorsque les spécifications du logiciel sont disponibles, l'analyste peut utiliser des techniques de raisonnement automatisés, comme, par exemple, la technique de la vérification de modèle. Dans cette thèse, nous avons utilisé la technique de la vérification de modèle pour l'analyse de la sécurité de OpenID et SAML SSO. Notre analyse a également examiné les options et configurations possibles de protocoles. Nos études ont conduit à la découverte d'une logique de vulnérabilité jusqu'alors inconnue. Cette vulnérabilité pourrait être utilisé par un attaquant d'exploiter l'authentification d'un utilisateur ou forcer l'utilisateur à accéder à une ressource sans son consentement explicite. Nous avons vérifié manuellement que la vulnérabilité existe dans les implémentations des protocoles disponibles sur Internet. Nous avons testé trois implémentations de SAML SSO, et deux implémentations de OpenID. Nous avons découvert que les implémentations quatre sur cinq souffrent de vulnérabilités que nous avons découvert grace à la vérification de modèle. En outre, nous avons constaté que la vulnérabilité logique peut être utilisé comme une rampe de lancement pour les attaques contre les XSS services SAML de Google. Tous nos résultats ont été discutés avec les membres de l'organe de normalisation de SAML qui ont élaboré errata.

A.3.1 Formal Analysis

L'analyse formelle de SAML SSO et OpenID a été menée par la plateforme AVANTSSAR. Nous avons modélisé protocoles utilisant un langage formel appelé Aslan + +, un langage formel pour spécifier les architectures orientées services, les politiques de sécurité et de propriétés de sécurité. En outre, nous avons utilisé SATMC (un vérificateur de modèle basé sur le problème de satisfiabilité SAT) pour découvrir les violations de la propriété de l'authentification mutuelle.

SAML SSO SATMC découvert une attaque à SAML SSO prouver qu'il ne pas satisfaire la propriété d'authentification mutuelle. L'attaque est représentée sur la Figure 1.

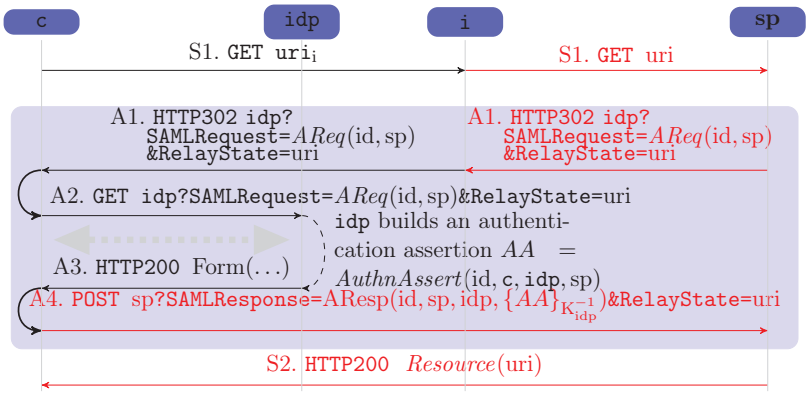


FIGURE 1 – Authentication Flaw of the SAML 2.0 Web Browser SSO Profile

L'attaque se déroule en quatre participants : un client, un IdP honnêtes, un SP honnête et SP malveillante. L'attaque est la suivante : C commence le protocole en demandant la ressource à SP malveillante. A ce stade, l'attaquant se fait passer pour C et nécessite une ressource différente de SP. SP se comporte selon le protocole et génère une demande d'authentification, qui est renvoyé à l'attaquant. Maintenant, l'attaquant répond au client par l'envoi d'une redirection vers l'IdP contenant $AReq(\mathrm{id}, \mathrm{sp})$ et uri au lieu $AReq(\mathrm{id_i}, \mathrm{i})$ et $\mathrm{uri_i}$ comme le protocole l'exige. Les étapes restantes sont effectuées selon le standard. L'attaque provoque le client de consommer une ressource de SP, mais le client à l'origine demandé des ressources à la SP malveillante.

OpenID SATMC découvert une attaque à OpenID prouver qu'il ne pas satisfaire la propriété d'authentification mutuelle. L'attaque est représentée sur la Figure 2

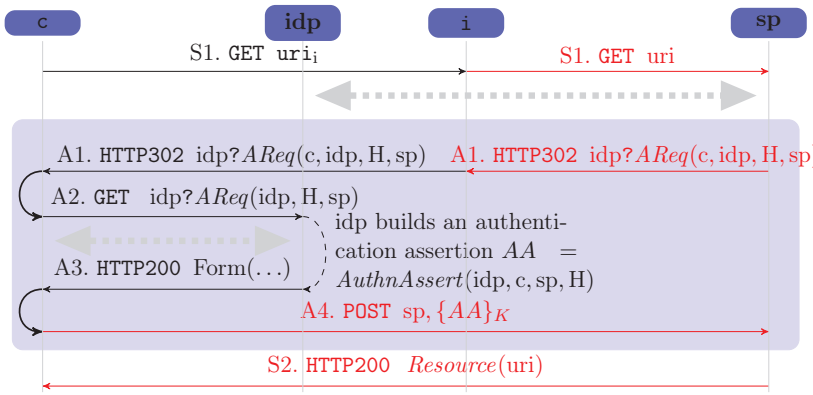


FIGURE 2 – Authentication Flaw of the OpenID SSO Protocol

A.4 From Model Checking to Security Testing

Dans le section A.3, nous avons montré que lorsque les spécifications du logiciel sont disponibles, la technique de la vérification de modèle peut être utilisé pour la découverte de vulnérabilités dans la logique. Cependant, les attaques détectées par le vérificateur de modèle démontrent la présence d'une vulnérabilité dans les modèles qui ne sont pas nécessairement reflété dans une vulnérabilité de logiciel. En outre, le vérificateur de modèle n'offre pas de support pour les implémentations de test. Par conséquent, les attaques sont normalement interprétées et exécutées sur le système réel manuellement. Dans ce section, nous proposons une technique pour les tests automatisés qui est entraîné par la vérification de modèle, et est en mesure de vérifier la présence de vulnérabilités dans les implémentations de protocoles de sécurité. Nous avons appliqué notre technique sur deux implémentations de SAML SSO, et deux implémentations de OpenID. Les expériences montrent que notre technique de test est capable de détecter les vulnérabilités SAML SSO et OpenID dans les systèmes réels.

Architecture Un aperçu de notre approche est illustrée à la Figure 3. Notre approche prend en entrée un modèle, une propriété de sécurité, et un

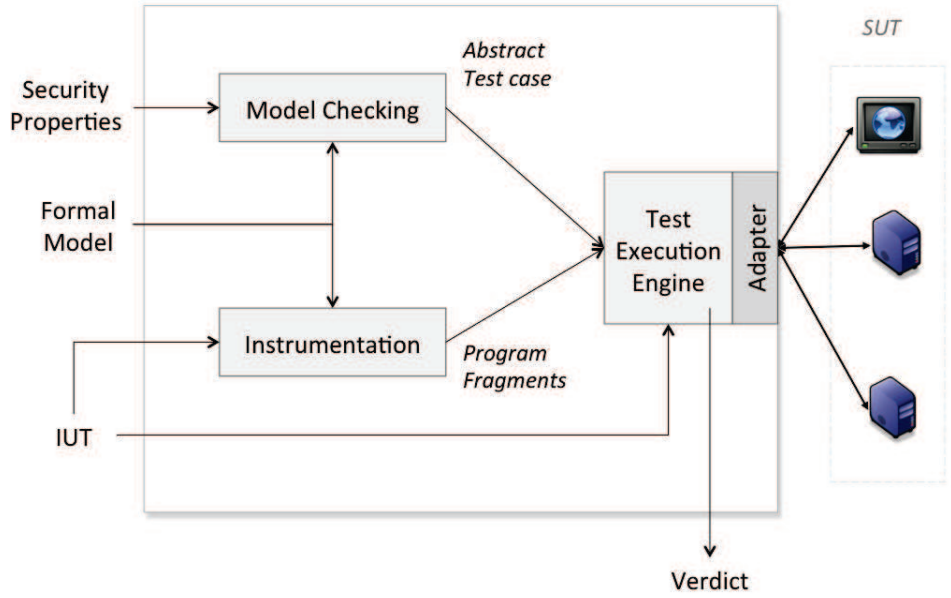


FIGURE 3 – Aperçu de notre approche

implementatione sous test (IUT). L'IUT est une structure de données qui contient une correspondance entre les symboles du modèle théorique et les valeurs réelles. En outre, il contient également les participants du protocole qui sont en cours de test. Notre approche comprend les étapes suivantes :

Model Checking A partir d'un modèle formel du protocole et une description des propriétés de sécurité prévu, le model checker explore systématiquement les états du modèle à constater des violations de propriété. Les violations sont identifiées par des contre-exemples qui sont utilisés comme des cas de test abstrait (*abstract test case*).

Instrumentation L'instrumentation calcule automatiquement et fournit l'exécuteur de tests (*Test Execution Engine*) un ensemble de fragments de programme qui codent la procédure de vérification (ou générer) les messages entrants (ou sortant) grâce à l'utilisation des fonctionnalités

offertes par l'adaptateur (Adapter) spécifiée dans l'IUT.

Execution L'exécuteur de tests (TEE) exécute des fragments de programme dans l'ordre établi par le cas de test abstrait. IUT spécifie ou les participants sont en cours de test (SUT) et qui, au contraire, seront simulées par TEE. Le verdetetto indique si le TEE a réussi à reproduire le scénario de test. Il est important de noter que si le verdict est négatif, notre approche peut être répété en exigeant le model checker un autre cas de test.

A.5 Black-Box Detection of Logic Flaws

Dans les sections précédents, nous avons vu que, à partir de la spécification formelle d'un protocole, il est possible d'automatiser le test de la sécurité d'une application réelle. Cependant, les spécifications qui décrivent l'évolution de l'état interne et le comportement attendu de l'utilisateur dans les applications Web, sont rarement disponibles. Le manque de documentation augmente la difficulté de la découverte de vulnérabilités logiques. Dans ce section, on propose une technique pour la découverte de vulnérabilité logique dans le cas ou les spécifications du logiciel ne sont pas disponibles. Nous avons appliqué notre technique sur sept applications de commerce électronique, exécutant plus de 3100 cas de tests, dont 900 ont violé le comportement attendu. Nos tests ont découvert 10 vulnérabilités logiques inconnus, dont cinq auraient permis à un attaquant de payer un montant inférieur ou même d'acheter en ligne gratuitement.

Le "OWASP Testing Guide v.3.0" propose une approche composée de 4 étapes manuelles pour vérifier la présence de vulnérabilités logiques dans les applications web en tenant compte de l'application comme une «boîte noire». Tout d'abord, le testeur étudie l'application à travers l'exploration des pages, et lisant la documentation disponible (comme l'aide en ligne). Par la suite, le testeur prépare les informations nécessaires à la conception des tests, y compris le flux de travail et le flux de données que le testeur a observé

dans la première phase. Après cela, le testeur procède à la conception des tests. Par exemple, les testeurs ont créé les tests qui subvertissent l'ordre de quelques étapes de de l'application, ou des tests qui permettent d'éviter certaines étapes de de l'application. Enfin, le testeur prépare l'environnement de test, exécute les tests, et évalue le résultat.

L'approche présentée dans ce section vise à automatiser les étapes ci-dessus dans un seul outil pour les tests de boîte noire. Tout d'abord, l'approche, à partir d'un ensemble des conversations HTTP, déduit un modèle de l'application en regroupant les ressources HTTP qui se réfèrent à la même "étape" dans le flux de travail. Ensuite, notre technique analyse le modèle et en extrait un ensemble de comportements liés à flux de travail et l'application de flux de données. Après, notre approche utilise un ensemble de schémas d'attaque pour générer les cas de test. L'approche se termine par l'exécution de cas de test contre une application web et utilise un oracle de décider si la logique de l'application a été violé.

Dans cette section, nous nous sommes concentrés sur un ensemble d'actions que l'attaquant peut exécuter contre l'application : Répétez les opérations, éviter les opérations, renverser l'ordre des opérations, et, enfin, d'échanger des valeurs entre les sessions utilisateurs. Pour chaque type d'attaque, nous avons conçu un schéma d'attaque. Un exemple de ces schémas d'attaque sont présentées dans la Figure 4.

Nous avons effectué des tests sur des applications de commerce électronique indiquées dans le Tableau 1. Notre outil peut également être utilisé pour tester l'application de commerce électronique en ligne (comme la boutique Amazon). Cependant, ces tests n'auraient pas été éthique et pourraient nuire à Amazon, les concessionnaires et les clients.

L'application de nos modes de schémas d'attaque extraites des traces d'entrée, notre outil a généré environ 3100 cas de test, une moyenne de 262 par application. Le nombre de cas de test générés pour l'application est résumée dans le Tableau 2.

6,6% de nos cas de test qui ont été exécutées avec succès correspond aux vulnérabilités. Ces cas ont été confirmés manuellement par l'inspection des registres du marchand et l'acheteur. 93,4% des cas de test ont trouvé

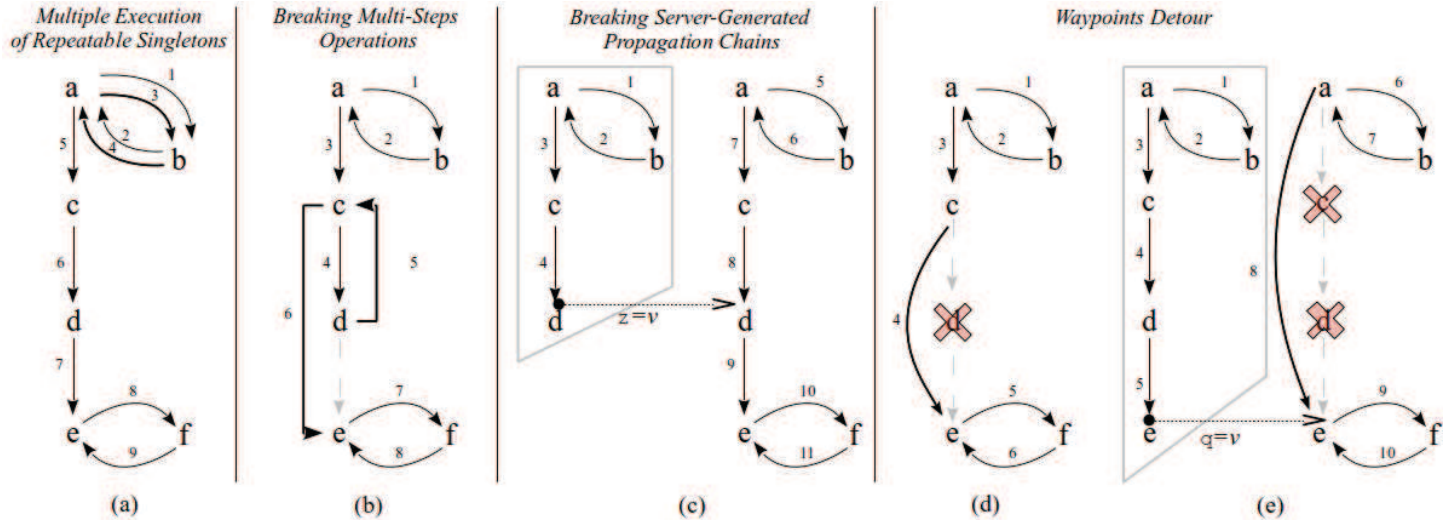


FIGURE 4 – Test case generation patterns

TABLE 1 – Popularity index

WebApp	Installations	WebApp	Installations
OpenCart	9,710,000	TomatoCart	119,000
Magento	3,130,000	osCommerce	80,500
PrestaShop	650,000	AbanteCart	21,200
CS-Cart	260,000		
		Total	13,970,700

des bugs dans la mise en oeuvre d'applications web. Dans tous ces cas, les tests ont été effectués jusqu'à la dernière ressource contenant un message de félicitations pour l'achat de produits.

A.6 Migration to SAP

Cette thèse a été principalement développée dans les laboratoires de SAP. Cela m'a permis de équilibrer le développement de nouvelles techniques d'analyse de leur application de scénarios industriels modernes. D'une part, j'ai appliqué les techniques de tests de protocoles de sécurité et des applications Web réelles découverte de nouvelles vulnérabilités logique. Deuxièmement, les résultats ont migré vers SAP afin de soutenir les ingénieurs de SAP dans (i) d'analyser la sécurité des configurations de protocoles de sécurité et (ii) de tester les implémentations pour trouver des vulnérabilités logiques.

SAP NetWeaver New Generation Single Sign-On SAP NetWeaver New Generation Single Sign-On (ci-apres NGSSO) met en oeuvre les principaux flux de SAML SSO, ses options, l'utilisation de SSL/TLS, et le décodage de messages cryptés et vérification de la signature digitale. En outre, les ingénieurs ont considéré SAP autres caractéristiques et les écarts par rapport aux spécifications du protocole. Dans ce section, nous allons décrire brièvement deux de ces.

La première différence par rapport à SAML SSO est l'utilisation de SPs sans état. SAML SSO fournit que les SPs convient de vérifier si l'ID des

TABLE 2 – Test case generation and execution

		Generation					Execution			
WebApp		Time	Repeat	Detour	MSteps	PChains	Time	Exec.	Not Exec.	Total
AbanteCart	Std	$\ll$ 0 :01	9	152	51	21	4 :51	74	159	233
Magento	Exp	0 :02	10	246	82	5	16 :23	240	103	343
	Std	0 :02	14	303	62	7	14 :50	210	176	386
OpenCart	Exp	0 :01	10	83	77	3	2 :34	140	33	173
	Std	0 :01	15	60	38	22	2 :08	71	64	135
osCommerce	Exp	$\ll$ 0 :01	4	142	13	6	3 :22	117	48	165
	Std	0 :01	8	144	63	10	3 :42	128	97	225
PrestaShop	Exp	$\ll$ 0 :01	12	100	22	3	2 :42	85	52	137
TomatoCart	Exp	0 :02	9	215	68	10	4 :54	238	64	302
	Std	0 :02	17	138	32	37	4 :36	115	109	224
CS-Cart	Exp	0 :05	8	562	24	6	12 :02	347	253	600
	Std	0 :02	16	137	54	15	5 :29	127	95	222
Total			132	2282	586	145		1892	1253	3145

TABLE 3 – Results

WebApp		Viol.	Bugs	Vuln.
AbanteCart	Std	17	16	1
Magento	Exp	65	65	-
	Std	126	126	-
OpenCart	Exp	58	46	12
	Std	30	30	-
osCommerce	Exp	42	22	20
	Std	35	34	1
PrestaShop	Exp	-	-	-
TomatoCart	Exp	90	65	25
	Std	24	24	-
CS-Cart	Exp	313	313	-
	Std	109	108	1
Total		909	849	60
		100%	93.4%	6.6%

réponses sont égaux à l'ID de la demande. Par conséquent, les SPs maintenir une table interne où stocker ces informations. Ces types de SPs sont appelés SPs qui ont l'état interne. Toutefois, les SPs avec l'etat peuvent être vulnérables à des attaques par déni de service dans laquelle un attaquant peut consommer la mémoire de la SPs envoyant des requêtes faux. Par conséquent, dans certains scénarios, les SPs qui sont sans état interne, sont préférés en raison de leur résistance à ces types d'attaques.

La deuxième différence est le contrôle des cookies de session au cours de l'exécution du protocole. SAML SSO ne nécessite pas l'utilisation des cookies dans aucune partie du protocole. Cependant, la SP peut utiliser des cookies pour mettre en œuvre des politiques particulières. Par exemple, le SP voudrait s'assurer que le navigateur Web qui transmet la réponse SAML est le même navigateur Web qui transmet les demandes SAML.

A Formal Analysis and Security Testing Tool Nous avons développé un outil qui met en oeuvre les techniques de vérification et de test montré dans les section A.3 et section A.4. En outre, nous avons étendu

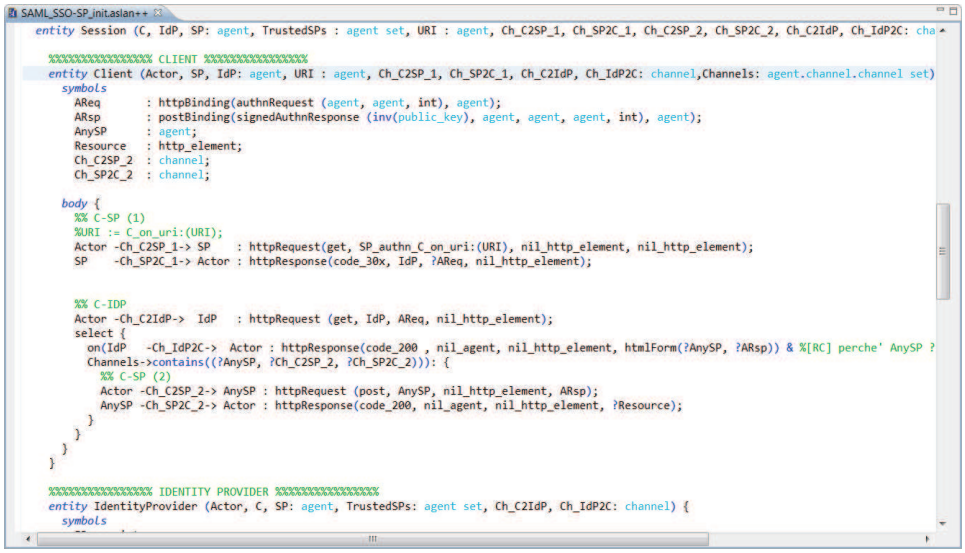


FIGURE 5 – ASLan++ Editor

ces techniques pour aider les ingénieurs d'analyser et de tester différentes configurations du même protocole.

A.7 Conclusions

L'état de l'art des techniques d'analyse ne fournit pas de support pour la découverte automatique des vulnérabilités dans la logique de l'entreprise de logiciels. Dans cette thèse, nous avons développé des techniques d'analyse pour résoudre les problèmes de ces technologies afin de permettre la découverte automatique des vulnérabilités logiques.

Dans la section A.3, nous avons montré une application de la technique de la vérification de modèle de protocoles de sécurité SSO SAML et OpenID. A partir des spécifications des protocoles, nous avons écrit des spécifications formelles qui capturent le comportement des participants, la structure des messages, et la composition des participants. Nous avons montré que la vérification de modèle peut détecter automatiquement les vulnérabilités dans

la logique des protocoles. Cependant, les résultats ne sont pas directement applicables aux implémentations. Nos résultats sont, cependant, été discutés avec les membres de l'organisation OASIS. En conséquence, OASIS a publié un erratum pour SAML.

Dans la section A.4, nous avons abordé le premier objectif de cette thèse, à savoir exécuter des tests contre les implémentations à partir des attaques détectées par le vérificateur de modèle. Nous avons proposé une approche qui comble le fossé entre les modèles formels et système réel grace à l'instrumentation du modele. L'instrumentation du modele consiste à calculer automatiquement un ensemble de fragments de programmes qui codent pour la génération et la vérification de messages. Les fragments sont finalement exécutés sur la base de l'attaque identifié par le vérificateur de modèle.

Les techniques des section A.3 et section A.4 ne sont applicables que lorsque les spécifications sont disponibles. Dans la section A.5, nous avons proposé une technique automatique pour le modèle de boîte noire qui ne nécessite pas d'apport. Notre approche en déduit un modèle de conversations HTTP. Par la suite, le modèle est utilisé pour générer les cas de test en fonction d'un certain nombre de modes d'attaque. Enfin, les tests sont effectués contre l'application Web, et un oracle décide si la logique de l'application a été violé.

Cette thèse a été développée principalement dans un contexte industriel. Les techniques présentées dans la section A.3 et section A.4 ont été intégrés dans un outil industriel, alors que la technique de la section A.5 est une preuve de concept. L'outil industriel a été utilisé pour tester quatre implémentations de SSO SAML et OpenID. En outre, cet outil a été utilisé dans SAP pour évaluer la sécurité des protocoles de sécurité mis en place par SAP. Le deuxième outil a été utilisé pour tester 12 application de commerce électronique Web, qui a découvert 10 vulnérabilités uniques logique, et 900 bugs. Tous les vulnérabilités critiques ont été signalées aux développeurs.