Network of Excellence

Deliverable D9.4

Enhanced Set of Solutions for Security Assurance for Services
<table>
<thead>
<tr>
<th><strong>Deliverable Number</strong></th>
<th>D9.4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Title of Deliverable</strong></td>
<td>Enhanced Set of Solutions for Security Assurance for Services</td>
</tr>
<tr>
<td><strong>Nature of Deliverable</strong></td>
<td>R</td>
</tr>
<tr>
<td><strong>Dissemination Level</strong></td>
<td>Public</td>
</tr>
<tr>
<td><strong>Internal Version Number</strong></td>
<td>3.1 (SVN Revision No. 4686)</td>
</tr>
<tr>
<td><strong>Contractual Delivery Date</strong></td>
<td>September 30, 2013</td>
</tr>
<tr>
<td><strong>Actual Delivery Date</strong></td>
<td>October 31, 2013</td>
</tr>
<tr>
<td><strong>Contributing WPs</strong></td>
<td>WP 9</td>
</tr>
<tr>
<td><strong>Editor(s)</strong></td>
<td>Christoph Sprenger (ETH Zurich)</td>
</tr>
<tr>
<td><strong>Author(s)</strong></td>
<td>Marianne Busch (LMU)</td>
</tr>
<tr>
<td></td>
<td>Walid Belkhir (INRIA)</td>
</tr>
<tr>
<td></td>
<td>Antonia Bertolino (CNR)</td>
</tr>
<tr>
<td></td>
<td>Manuel Clavel (IMDEA)</td>
</tr>
<tr>
<td></td>
<td>Frédéric Dadeau (INRIA)</td>
</tr>
<tr>
<td></td>
<td>Said Daoudagh (CNR)</td>
</tr>
<tr>
<td></td>
<td>Carolina Dania (IMDEA)</td>
</tr>
<tr>
<td></td>
<td>Donia El Kateb (UNI-LUX)</td>
</tr>
<tr>
<td></td>
<td>Christopher Henard (UNI-LUX)</td>
</tr>
<tr>
<td></td>
<td>Erwin Hess (SIEMENS)</td>
</tr>
<tr>
<td></td>
<td>Nora Koch (LMU)</td>
</tr>
<tr>
<td></td>
<td>Yves Le Traon (UNI-LUX)</td>
</tr>
<tr>
<td></td>
<td>Francesca Lonetti (CNR)</td>
</tr>
<tr>
<td></td>
<td>Eda Marchetti (CN)</td>
</tr>
<tr>
<td></td>
<td>Ilaria Matteucci (CN)</td>
</tr>
<tr>
<td></td>
<td>Tejeddin Mouelhi(UNI-LUX)</td>
</tr>
<tr>
<td></td>
<td>Binh Thanh Nguyen (ETH Zurich)</td>
</tr>
<tr>
<td></td>
<td>Viet Hung Nguyen (UNITN)</td>
</tr>
<tr>
<td></td>
<td>Martin Ochoa (SIEMENS)</td>
</tr>
<tr>
<td></td>
<td>Mike Papadakis (UNI-LUX)</td>
</tr>
<tr>
<td></td>
<td>Michael Rusinowitch (INRIA)</td>
</tr>
<tr>
<td></td>
<td>Riccardo Scandariato (KUL)</td>
</tr>
<tr>
<td></td>
<td>Christoph Sprenger (ETH Zurich)</td>
</tr>
<tr>
<td></td>
<td>Mohammad Torabi Dashti (ETH Zurich)</td>
</tr>
<tr>
<td></td>
<td>Artsiom Yautsiukthin (CR)</td>
</tr>
<tr>
<td><strong>Reviewer(s)</strong></td>
<td>Juan Caballero (IMDEA)</td>
</tr>
<tr>
<td></td>
<td>Ketil Stelen (SINTEF)</td>
</tr>
<tr>
<td></td>
<td>Marinella Petrocchi (CN)</td>
</tr>
</tbody>
</table>
Abstract

The main objective of Work Package 9, which this deliverable reports on, is to enable assurance in the development of software-based services in order to increase confidence in their security. The core goal is to incept a transverse methodology that enables to manage assurance throughout the software development life cycle (SDLC).

In this deliverable, we report on the further development of our methods and tools for security assurance for services that we have obtained during the third year of the NESSoS project. In early assurance at the level of requirements, architecture, and design using techniques, we have several contributions concerning model extraction from UML-based design models and algorithmic verification of security protocols. In the complementary implementation-based assurance techniques, we present our research results in the testing of security protocols and access control policies as well as in the area of runtime verification. We record very good overall progress, both in terms of the methods and of the tools developed. The third year is also marked by ongoing and new collaborations which are fostered by the NoE and have resulted in joint publications.

Keyword List

Service assurance, model extraction, refinement, abstraction, security protocols, access control policies, testing, vulnerability testing, model-based testing, runtime verification, quantitative security.
### Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Type of Change</th>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Initial setup and ETH contribution to Task 9.1</td>
<td>Christoph Sprenger (ETH)</td>
</tr>
<tr>
<td>0.2</td>
<td>KUL contribution to Task 9.4</td>
<td>Riccardo Scandariato (KUL)</td>
</tr>
<tr>
<td>0.3</td>
<td>INRIA contributions to Task 9.2</td>
<td>Michael Rusinowitch (INRIA)</td>
</tr>
<tr>
<td>0.4</td>
<td>SIEMENS-ETH contribution to Task 9.2</td>
<td>Mohammad Torabi Dashti (ETH)</td>
</tr>
<tr>
<td>0.5</td>
<td>SIEMENS contribution to Task 9.1, Cryptography</td>
<td>Erwin Hess (Siemens)</td>
</tr>
<tr>
<td>0.6</td>
<td>LMU-CNR contribution to Task 9.1, X-CREATE toolchain</td>
<td>Marianne Busch (LMU)</td>
</tr>
<tr>
<td>0.7</td>
<td>IMDEA contribution to Task 9.1, model analysis</td>
<td>Manuel Clavel (IMDEA)</td>
</tr>
<tr>
<td>0.8</td>
<td>UNI-LUX-CNR contribution to Task 9.2, test case prioritization</td>
<td>Eda Marchetti (CNR)</td>
</tr>
<tr>
<td>0.9</td>
<td>LMU contribution to Task 9.1, TextualUWE</td>
<td>Marianne Busch (LMU)</td>
</tr>
<tr>
<td>1.0</td>
<td>Complete draft</td>
<td>see page 3</td>
</tr>
<tr>
<td>2.0</td>
<td>Ready for internal review</td>
<td>see page 3</td>
</tr>
<tr>
<td>3.0</td>
<td>Pre-final version</td>
<td>see page 3</td>
</tr>
<tr>
<td>3.1</td>
<td>Final version, approved by reviewers</td>
<td>Christoph Sprenger (ETH)</td>
</tr>
</tbody>
</table>
# Table of Contents

**List of Figures** ........................................................................................................... 9

**List of Tables** ............................................................................................................. 11

1 **Introduction** ............................................................................................................. 15

2 **Early Assurance** ...................................................................................................... 17
   2.1 Mapping security-design models to enable formal analysis ................................. 17
       2.1.1 Analysis of ActionGUI models .................................................................. 17
       2.1.2 Verification of security features in UWE models ................................. 19
       2.1.3 A toolchain for designing and testing XACML policies ................. 22
   2.2 Algorithmic verification ....................................................................................... 25
       2.2.1 Sound security protocol transformations ............................................. 25
   2.3 Cryptography ....................................................................................................... 26
       2.3.1 Quantum-computing-resistant cryptographic algorithms .................... 26

3 **Assurance in Implementation** ............................................................................. 29
   3.1 Testing ................................................................................................................... 29
       3.1.1 Mutation-based testing of security protocols .................................... 29
       3.1.2 VERA: A flexible model-based vulnerability testing tool .............. 31
       3.1.3 Test case prioritization for XACML policies ................................... 33
       3.1.4 Testing of PolPA-based usage control systems ................................ 36
   3.2 Runtime verification ............................................................................................. 40
       3.2.1 Quantitative controllers specified with generalized process algebras ...... 40
       3.2.2 Synthesizing a mediator satisfying policy constraints ....................... 43

4 **Quantitative Methods** ......................................................................................... 47
   4.1 Vulnerability prediction in Firefox ..................................................................... 47
   4.2 Modelling adaptive attacker’s behaviour ....................................................... 50

5 **Relations to Other Work Packages** ..................................................................... 55

6 **Conclusions** .......................................................................................................... 57

A **NESSoS WP 9 Third-year Publications** .............................................................. 59

**Bibliography** .............................................................................................................. 61
List of Figures

Figure 2.1: UWE: bonus application example ................................................. 21
Figure 2.2: Toolchain ................................................................................. 24
Figure 3.1: Platform Architecture ............................................................. 30
Figure 3.2: PayPal Standard Workflow ...................................................... 31
Figure 3.3: General injection attacker model ............................................. 33
Figure 3.4: Testing Framework ................................................................. 39
Figure 3.5: Policy: no private messages between two clients ..................... 44
Figure 4.1: Future prediction in Firefox via text mining. The performance is stable over time. ...... 49
Figure 4.2: Example of a computer system (a) and the attack graph (b). ............. 52
List of Tables

Table 3.1: Description of the six policies ..................................................... 36
Table 3.2: Semantic rules for quantitative control operators. ............................... 42
Table 4.1: Predicting future Firefox versions. The model produces excellent performance results that outperform the state of the art. ............................................................. 48
Executive Summary

This deliverable reports on the results obtained in WP 9 during the third year of the NESSoS project. Our results build on the state-of-the-art in security assurance as described in the NESSoS Deliverable ID9.1 [82] and the results obtained during the first and second year as reported in Deliverables D9.2 [81] and D9.3 [83]. Here is a brief overview of our third-year results.

Mapping security-design models to enable formal analysis (Section 2.1) We report on three lines of work in this area. First, we have developed the OCL2FOL+ tool for analyzing ActionGUI models and prove that Boolean OCL expressions specifying authorization constraints, conditions, or objects never evaluate to null or invalid values at run-time. Second, we report on TextualUWE, a textual notation and verification algorithms for UWE models. The goal was to provide a textual alternative to the graphical UWE notation, which is easy to read for humans and exploitable by algorithms. Third, we present an integrated framework not only for designing security policies, but also for testing the compliance of the derived XACML sources with the initial models.

Algorithmic verification (Section 2.2) We have identified a class of security protocol transformations, which can be used to develop (families of) security protocols by refinement or to abstract existing protocols in order to increase the efficiency of verification tools. We prove the soundness of these transformations with respect to an expressive security property specification language.

Cryptography (Section 2.3) We have investigated the potential consequences of the availability of a quantum computer for cryptography and to give – from today’s point of view – an overview on cryptographic algorithms resistant against quantum computing.

Testing and debugging (Section 3.1) Our work on testing and debugging has focused on four topics. First, we improve previously proposed black-box testing methods to semi-automatically discover new attacks and vulnerabilities. We propose an architecture for converting abstract attacks obtained by mutation techniques to concrete test scenarios that are executable on implementations. Second, we propose VERA, a flexible tool for model-based vulnerability testing. In contrast to many existing tools, VERA allows users to define attacker models where the payloads and the behavior are cleanly separated and that abstract away from low-level implementation details. Third, we propose a new approach to access control test prioritization that relies on similarity metrics. Fourth, in order to prevent security compromises due to errors in the authorization system, we propose two testing strategies specifically conceived for validating the history-based access control and the continuous usage control functionalities of the PolPA Policy Decision Point (PDP).

Runtime verification (Section 3.2) In our work on system compliance and policy enforcement, we have concentrated on two aspects. First, we propose a formal framework for the quantitative evaluation of enforcement strategies, extending the notion of controller processes with weights on transitions, valued in a semiring. Second, we have worked on the synthesis of secure mediators. Given a client specification, an available community of services, and a policy, we propose a new, more expressive approach to synthesize a mediator that suitably directs the actions of the client to the services whilst respecting the policy.

Quantitative methods (Section 4) In this area, we can report on progress in two directions. First, we validate our vulnerability prediction technique based on text mining. We applied the technique in a different class of applications (web browser, namely Firefox), a different programming language (C/C++) and using vulnerability data coming from authoritative sources (NVD/MFSA). Second, we present a model for the behaviour of an attacker with uncertain knowledge about a computer system. The proposed model allows a finer-grained analysis of the security of computer systems and, thus, leads to a more accurate assessment of possible risks.

The methods and tools presented here cover all the major activities related to providing assurance in the design and implementation phases of the SDLC. The third year was also marked by the emergence of a number of new collaborations between project and associated partners as well as the deepening of existing collaborations.
1 Introduction

This deliverable reports on the results obtained in WP 9 during the third year of the NESSoS project. Our results build on the state-of-the-art in security assurance as described in the NESSoS Deliverable ID9.1 [82] and the results obtained during the first and second year as reported in Deliverables D9.2 [81] and D9.3 [83]. In the following, we give an overview of our third-year results.

Mapping security-design models to enable formal analysis (Section 2.1) We can report on three lines of work in this area.

First, we have developed the OCL2FOL+ tool to address the problem of coping with undefinedness in OCL while at the same time being able to use SMT solvers (e.g., Z3 and Yices) to automatically carry out OCL reasoning. We plan now to use OCL2FOL+ for analyzing ActionGUI models. In particular, we are interested in proving that Boolean OCL expressions specifying authorization constraints (in permissions), conditions (in if-then-else statements), or objects (in actions) will never evaluate to null or invalid values at run-time.

Second, we report on TextualUWE, a textual notation and verification algorithms for UWE models. The goal was to provide a textual alternative to the graphical UWE notation, which is easy to read for humans and exploitable by algorithms. We have implemented the transformation from the graphical to the textual UWE notation. Additionally, we are working on algorithms to check security features of TextualUWE models, such as the reachability of certain parts of a web application by a user in a given role.

Third, we present an integrated framework not only for designing security policies, but also for testing the compliance of the derived XACML sources with the initial models. Inspired by the conformance testing process, we use some XACML-based testing strategies for generating appropriate test cases enabling the testing of functional aspects, constraints, permissions, and prohibitions.

Algorithmic verification (Section 2.2) We have focused on identifying a class of security protocol transformations, which can be used to develop (families of) security protocols by refinement or to abstract existing protocols to increase the efficiency of verification tools. We prove the soundness of these transformations with respect to an expressive security property specification language covering secrecy and authentication properties.

Cryptography (Section 2.3) We have investigated the potential consequences of the availability of a quantum computer for cryptography and to give – from today’s point of view – an overview on cryptographic algorithms resistant against quantum computing.

Testing and debugging (Section 3.1) Our work on testing and debugging has focused on four topics.

First, we improve previously proposed black-box testing methods in order to semi-automatically discover new attacks and vulnerabilities. This contribution proposes an architecture for converting abstract attacks, obtained automatically by mutation techniques on protocol specifications, to concrete test scenarios that are directly executable on protocol implementations.

Second, we propose VERA, a flexible tool for model-based vulnerability testing. In contrast to many existing tools, VERA allows users to define attacker models where the payloads and the behavior are cleanly separated and that abstract away from low-level implementation details such as HTTP requests.

Third, in order to ensure that security policies are correctly enforced, it is important to identify potential security flaws and bugs by an efficient testing process. However, in practice, exhaustive testing is impossible due to budget constraints. It is therefore required to prioritize tests to run the most important and relevant subset of tests. We propose a new approach to access control test prioritization that relies on similarity metrics.

Fourth, in order to prevent security compromises due to errors in the authorization system, we propose two different testing strategies specifically conceived for validating the history-based access control and the continuous usage control functionalities of the PolPA Policy Decision Point (PDP). The former is based on a fault model able to highlight the problems and vulnerabilities that could
occur during the PDP implementation. The latter combines the standard technique for conditions coverage with a methodology for simulating the continuous control of the PDP during the runtime execution.

Runtime verification (Section 3.2) In our work on system compliance and policy enforcement, we have concentrated on two aspects.

First, in order to enforce a given security policy, a security enforcement mechanism runs in parallel with a system to check and modify its run-time behaviour. For each policy, several enforcement strategies are possible, usually reflecting trade-offs one has to make to satisfy the policy. To evaluate them, multiple dimensions, such as security, cost of implementation, or cost of attack, must be taken into account. We propose a formal framework for the quantification of enforcement strategies, extending the notion of controller processes (mimicking the well-known edit automata) with weights on transitions, valued in a semiring.

Second, we have worked on the synthesis of secure mediators. Given a client specification, an available community of services, and a policy, we propose a new approach to synthesize a mediator that suitably directs the actions of the client to the services whilst respecting the policy. The associated composition synthesis problem can be solved for communicating guarded variable automata, a class of automata operating on infinite alphabets.

Quantitative methods (Section 4) In this area, we can report on progress in two directions.

First, we validate our vulnerability prediction technique based on text mining. We applied the technique in a different class of applications (web browser, namely Firefox), a different programming language (C/C++) and using vulnerability data coming from authoritative sources (NVD/MFSA). This work confirms that our technique works very well and its performance outclasses the state of the art. The technique could be used to identify the components of an application that deserve special attention from the quality assurance team, e.g., to prioritize the testing plans and the code review activities.

Second, we describe a model for the behaviour of an attacker with uncertain knowledge about a computer system. In this model, the attacker tries different attack paths if the initially selected ones cannot be completed. The proposed model allows a finer-grained analysis of the security of computer systems and, thus, leads to a more accurate assessment of possible risks.

The remaining sections of this deliverable are organized as follows. In Section 5 we discuss relationships to other NESSoS work packages. We draw conclusions and provide an outlook on future work in Section 6. Finally, in Appendix A, we list the publications produced in this work package during the third year of the project.
2 Early Assurance

Early detection of security failures in Future Internet applications reduces development costs and improves assurance in the final system. In this section, we report on our results for early assurance. In Section 2.1, we present formal mappings from security design models to other formalisms for which automated or semi-automated analysis tools are available. In Section 2.2 is devoted to algorithmic verification techniques, in particular to abstraction and refinement of security protocols. Section 2.3 investigates the available cryptographic options under the scenario of quantum computers becoming available.

2.1 Mapping security-design models to enable formal analysis

2.1.1 Analysis of ActionGUI models

In Deliverable D9.3 we proposed a methodology for proving properties of ActionGUI [1] models. Our methodology is based on the formalization of these properties in OCL [66]. Then, we will translate the corresponding OCL expressions into first-order logic and use SMT solvers (e.g., Z3 [32] or Yices [36]) to prove their logical properties.

At present, however, the OCL language includes two constants, null and invalid, to represent undefinedness. This effectively turns OCL into a four-valued logic. It makes also problematic its mapping to first-order logic and, as a consequence, hinders the use of SMT solvers for OCL reasoning. In [29] we have addressed this problem and proposed a solution to cope with undefinedness in OCL which we summarize in this section.

Introduction

In the past decade, there has been a plethora of proposals to map OCL into different formalisms for which reasoning tools may be readily available. Most of these proposals have not considered the parts of the OCL language that deal with undefinedness or generate it; notable exceptions to this are [20,21] and [3], although the latter only deal with null-related undefinedness.

In [27] we proposed a mapping from OCL to first-order logic, and we implemented it in a tool called OCL2FOL. This mapping supports the direct use of SMT solvers to check the satisfiability of OCL expressions, but under specific restrictions to avoid the problem of undefinedness in OCL. After experimenting with OCL2FOL, we have come to realize that not coping with undefinedness imposes severe limitations on the applicability of the tool. To overcome these limitations, we have recently proposed OCL2FOL+ [29].

OCL2FOL

O2F+ is a mapping from a subset of OCL into first-order logic (FOL), such that the following holds: Let \( M \) be a data model, let \( I \) be an instance of \( M \) (where all attributes’ values are defined) and let \( expr \) be a Boolean OCL expression (within the O2F+’s range), whose context is also \( M \). Then,

\[
\text{Eval}(expr, I) = \text{true} \iff O2F_{\text{env}}(M, I, expr) = O2F^+(expr)
\]

where \( \text{Eval}(expr, I) \) is the value returned by the evaluation of the expression \( expr \) in the scenario \( I \), and \( O2F_{\text{env}}(M, I, expr) \) is the union of the set of formulas resulting from calling three auxiliary mappings, namely, \( O2F_{\text{data}} \), \( O2F_{\text{inst}} \), and \( O2F_{\text{def}} \), on, respectively, the data model \( M \), the scenario \( I \), and the expression \( expr \). That is,

\[
O2F_{\text{env}}(M, I, expr) = O2F_{\text{data}}(M) \cup O2F_{\text{inst}}(I) \cup O2F_{\text{def}}(expr)
\]

Basically, \( O2F_{\text{data}}(M) \) and \( O2F_{\text{inst}}(I) \) formalize in first-order logic the data model \( M \) and the scenario \( I \). Then, \( O2F_{\text{def}}(expr) \) defines in first-order logic the collections (if any) that are referred to by subexpressions in \( expr \). Finally, \( O2F^+(expr) \) formalizes in first-order logic the meaning of OCL expression \( expr \) in the context of the data model \( M \) and the scenario \( I \). As expected, the map \( O2F^+ \) is defined recursively over the structure of OCL expressions.
OCL2FOL\textsuperscript{+}: Coping with undefinedness

In a nutshell, the problem of undefinedness in OCL, when trying to use existing first-order logic reasoning tools for OCL reasoning, is the problem of mapping a four-valued logic (true, false, null, and invalid) into a two-valued logic (true and false). In general, when evaluating a Boolean OCL expression in a scenario, the result can be either true, false, null or invalid. However, when interpreting a first-order formula in a model, the result can only be true or false.

To cope with undefinedness our solution consists on defining four different mappings, namely, \(O2F_{\text{true}}\), \(O2F_{\text{false}}\), \(O2F_{\text{null}}\), and \(O2F_{\text{invalid}}\), such that the following holds: Let \(M\) be a data model, let \(I\) be an instance of \(M\) (where now not all attributes’ values need to be defined), and let \(exp\) be a Boolean OCL expression whose context is \(M\) (within the \(O2F\)\textsuperscript{−}’s range, extended now to include expressions built with the type-casting operators, the division operator, and the operators any, oclIsUndefined, and oclIsInvalid). Then,

\[
\begin{align*}
\text{Eval}(\text{expr}, I) = \text{true} & \iff O2F_{\text{true}}^+(M, I, \text{expr}) \models O2F_{\text{true}}(\text{expr}) \\
\text{Eval}(\text{expr}, I) = \text{false} & \iff O2F_{\text{false}}^+(M, I, \text{expr}) \models O2F_{\text{false}}(\text{expr}) \\
\text{Eval}(\text{expr}, I) = \text{null} & \iff O2F_{\text{null}}^+(M, I, \text{expr}) \models O2F_{\text{null}}(\text{expr}) \\
\text{Eval}(\text{expr}, I) = \text{invalid} & \iff O2F_{\text{invalid}}^+(M, I, \text{expr}) \models O2F_{\text{invalid}}(\text{expr})
\end{align*}
\]

where \(\text{Eval}(\text{expr}, I)\) is the value returned by the evaluation of the expression \(\text{expr}\) in the scenario \(I\), and \(O2F_{\text{true}}^+(M, I, \text{expr})\) is the union of the set of formulas resulting from calling three auxiliary mappings, namely, \(O2F_{\text{data}}^+, O2F_{\text{inst}}^+, \) and \(O2F_{\text{def}}^+,\) on, respectively, the data model \(M\), the scenario \(I\), and the expression \(\text{expr}\). As their names suggest, these three mappings are extensions of the mappings \(O2F_{\text{data}}, O2F_{\text{inst}}, \) and \(O2F_{\text{def}}\). That is,

\[
O2F_{\text{true}}^+(M, I, \text{expr}) = O2F_{\text{data}}^+(M) \cup O2F_{\text{inst}}^+(I) \cup O2F_{\text{def}}^+(\text{expr})
\]

Tool support

We have implemented our four mappings in a tool, called OCL2FOL\textsuperscript{+} [2]. OCL2FOL\textsuperscript{+} takes as input a data model \(M\) (the context), a set of Boolean OCL expressions \(\text{exp}_1, \ldots, \text{exp}_n\) (the hypothesis), a Boolean OCL expression \(\text{exp}_{n+1}\) (the assertion), and one of the following keywords (the type): true, false, null, inval. From the given input, OCL2FOL\textsuperscript{+} automatically generates the following set of formulas (using SMT-Lib 2.0 syntax):

\[
O2F_{\text{data}}^+(M) \cup \bigcup_{i=1}^{n} O2F_{\text{true}}(\text{exp}_i) \cup \bigcup_{i=1}^{n+1} O2F_{\text{def}}(\text{exp}_i) \cup O2F_{\text{type}}(\text{exp}_{n+1})
\]

Conclusions

We have addressed the problem of coping with undefinedness in OCL while at the same time being able to use SMT solvers (e.g., Z3 and Vc++I) to automatically carry out OCL reasoning. Our solution follows the same principles underlying OCL2FOL, but consists now of four different mappings which formalize, respectively, when an expression evaluates to true, when to false, when to null, and when to invalid. Our solution also differs from [20, 21], since we pursue different goals. In particular, while we are able to support automated OCL reasoning, we can only cover a subset of OCL. On the other hand, the solution presented in [20, 21], although it covers the full OCL language, it can only provide interactive theorem-proving capabilities. As part of this work, we have also implemented our four mappings in a tool called OCL2FOL\textsuperscript{+} [2].

We plan to use OCL2FOL\textsuperscript{+} for analyzing ActionGUI models. In particular, we are interested in proving that Boolean OCL expressions specifying authorization constraints (in permissions), conditions (in if-then-else statements), or objects (in actions) will never evaluate to null or invalid at run-time. For this, we will apply the same methodology proposed in Deliverable D9.3, but using the mapping to first-order logic implemented in OCL2FOL\textsuperscript{+}. 
2.1.2 Verification of security features in UWE models

To secure web applications is increasingly important because of rising cybercrime as well as the growing awareness of data privacy. Besides confidential connections and authentication, both data access control and navigational access control are the most relevant security features in this field. However, adding such security features to already implemented web applications is an error-prone task. Therefore, the goal is to include security features in early stages of the development process, i.e., at requirements specification and design modeling level.

Existing approaches, such as OOHRIA [58], OOWS [87], WebML [60], UWE [24,45], or ActionGUI [8] already provide well-known methods and tools for the design of web applications. Most of them follow the principle of “separation of concerns” using separate models for different views on the application, such as e.g. content, navigation, presentation and business processes. Most of the available methods do not support modeling security-features, whereas the UWE approach by Koch et al. [24] and the ActionGUI approach by Basin et al. [8] define models for security features as access control and authorization. ActionGUI’s proprietary notation comprises data models, security models and GUI models, and the application logic is represented using OCL. UWE provides a set of UML stereotypes for each view, defined by a so-called UML profile. UWE’s main focus is on the process of discussing and planning an application from different points of view as e.g. requirements, content (data model), navigation, users and roles, basic access control rights, presentation and process.

However, graphical models are not always the means of choice, as some security engineers prefer to model in a textual way. It is possible to export UML element structures in a so-called XMI (XML Metadata Interchange) format, but the exported files are not human-readable due to many generated cross-links and due to their length. Simple models easily cover thousands of lines when exported in XMI. The complex and non-intuitive structure does not contribute to ease model checking or defining the model’s semantics, although tools exist to support such tasks, e.g., tools provided by the Eclipse Modeling Framework Project1.

In our modeling approach, we aim at providing a Domain Specific Language (DSL) for UWE, called TextualUWE. The focus is on creating an intelligible language, which is not only easy to read but also exploitable by algorithms. As we want to open the way to expressive algorithms, we decided to use Scala2, a multi-paradigm programming language which supports object-oriented programming in both functional and imperative style. For our work we use the functional style, as it allows writing short and precise algorithms to support TextualUWE. The data structure for our DSL is also plain Scala so that there is no need for a special DSL editor.

Existing UWE diagrams can automatically be transformed into TextualUWE. The other way around is also possible, however good diagram layout algorithms are hard to find. So far, we implemented the transformation from graphical UWE diagrams to TextualUWE. Additionally, we are working on algorithms to check security features of TextualUWE models, as e.g., which part of the web application can be reached by a user which is associated to a certain role. Further verifiable features are to find inconsistencies in the model or to check what happens when parts of pages (so-called navigation nodes) are illegally accessed.

In the following, we outline UML-based Web Engineering (UWE) [24], the security-aware engineering approach we have chosen for modeling web applications. Afterwards, we describe TextualUWE and its verification possibilities by example. Note that TextualUWE is currently work in progress.

UML-based web engineering – UWE

One of the cornerstones of the UWE language is the “separation of concerns” principle using separate models for different views. However, we can observe that security features are cross-cutting concerns which cannot be separated completely. The main UWE models are:

The Requirements Model defines (security) requirements for a project.

The Content Model contains the data structure used by the application.

The UWE Role Model describes a hierarchy of user groups to be used for authorization and access control issues. It is usually part of a User Model, which specifies basic structures, as e.g., that a user can take on certain roles simultaneously.

The Basic Rights Model describes access control policies. It constrains elements from the Content Model and from the Role Model.

The Presentation Model sketches the web application’s user interface.

The Navigation State Model defines the navigation flow of the application and navigational access control policies. The former shows which possibilities of navigation exist in a certain context. The latter specifies which roles are allowed to navigate to a specific state and the action taken in case access cannot be granted. In a web application such actions can be, e.g., to logout the user and to redirect to the login form or just to display an error message. Furthermore, secure connections are modeled.

For each view, an appropriate type of UML diagram is used, e.g., a state machine for the Navigational Model. In addition, the UWE Profile adds a set of stereotypes, tag definitions and constraints, which can be downloaded from the UWE website [49]. Stereotypes can then be applied to UML model elements and values can be assigned to tags, which are associated to a stereotype.

TextualUWE

The textual version of UWE comes as two Scala packages: one specifying the DSL and another one containing functional verification algorithms. As an example, we detail how UWE’s Navigation States model is described by TextualUWE.

As mentioned above, the Navigation States model is based on a UML state machine, which means we have to express state machines, states and transitions. In addition, stereotypes can be set on states and transitions. States can be “simple” or “composite”, where composite states contain state machines which are executed in parallel. The following listing shows an excerpt of our definition:

```scala
object NavigationStateMachine {
  sealed abstract trait State
  case class SimpleState(name: String,
    stereotypes: Set[StateStereotype] = Set()) extends State
  case class CompositeState(name: String,
    regions: Set[StateMachine],
    stereotypes: Set[StateStereotype] = Set()) extends State

  case class Transition(source: State,
    target: State,
    leftCStates: Int = 0,
    guard: String = "",
    stereotypes: Set[TransStereotype] = Set(),
    enteredCStates: List[CompositeState] = List())

  case class StateMachine(initialState: State, transitions: Set[Transition])
...
}
```

Transitions connect two states and additionally they record which composite states were entered and how many composite states were left by this transition (which is denoted by the parameter \texttt{leftCStates}). This is necessary due to the nesting of state machines within composite states.

Figure 2.1 depicts a UWE example of a simple smart grid application. The application, which is also used in deliverable D8.4, represents a prototype of an energy offer management including optional bonus handling. It provides two different user roles namely Provider and Customer: Providers manage and sell energy packages including optional bonus programs for customers. Customers have the possibility to buy offered energy packages and to get associated bonus codes. Details are omitted at this point, because they do not contribute to the understanding of TextualUWE. The textual version of our example is listed below:
Figure 2.1: UWE: bonus application example

object NavStateSmartGrid {
  val loginViaPF = SimpleState("LoginViaPasswordForm")
  val loginArea = CompositeState("LoginArea",
      Set(StateMachine(loginViaPF, Set())),
      Set(new NavigationalNode(true)))

  val customerArea = SimpleState("CustomerArea",
      Set(new Session(Set("customer")))), // refers to ... in diagram

  val providerHome = SimpleState("ProviderHome")
  val launchNewBonusProgram = SimpleState("LaunchNewBonusProgram")
  val innerTrans = Set(Transition(providerHome, launchNewBonusProgram),
      Transition(launchNewBonusProgram, providerHome))
  val providerArea = CompositeState("ProviderArea",
      Set(StateMachine(providerHome, innerTrans)),
      Set(new Session(Set("provider"))))
  val errorState = SimpleState("Error")

  val interTrans = Set(Transition(providerArea, errorState, 0, "unauthorized"),
                   Transition(customerArea, errorState, 0, "unauthorized"),
                   Transition(errorState, loginArea),
                   Transition(providerArea, loginArea),
                   Transition(customerArea, loginArea),
                   Transition(loginArea, providerArea, 0, "role = provider"),
                   Transition(loginArea, customerArea, 0, "role = customer"))
}
val ts = interTrans ++ innerTrans
val outerState = CompositeState("SmartGridBonusApplication",
  Set(StateMachine(loginArea, ts)),
  Set(new Session(Set(), ",", TTthsts)))
val sm = StateMachine(outerState, ts)
}

As one can see, it is a matter of taste, if the graphical or the textual version of UWE is preferred. To
be able to work with both versions, we implemented a transformation from the UWE to the TextualUWE
notation [73] using the Acceleo framework\(^3\). Acceleo is an open-source text generator which is implements
a template-based approach for extracting information from input models.

For the verification of model features, we use Scala functions. An example is the query which states
can be reached if the user takes on certain roles (or other conditions are met, which have to be defined
separately). The algorithm is simple: each state can be a simple state or a composite state. For simple
states, an inner function makes use of sentry functions to decide whether or not the state is accessible.
Sentry functions can, e.g., test for roles, or they can allow to enter all transitions and states, which results
in a list of all reachable states, regardless of the roles, a user plays. For complex states, each state
machine is examined.

Conclusions

To sum up, TextualUWE provides a textual notation and verification algorithms for UWE models. Fur-
thermore, a transformation from the graphical to the textual UWE notation is implemented. Future work
includes taking further web security features into account (cf. deliverable D7.4) and implementing more
validation algorithms. It is also interesting to specify semantics in the sense of defining what can happen
in each step when navigating through the web application. Especially the connections between several
UWE models are relevant at this point.

2.1.3 A toolchain for designing and testing XACML policies

In Deliverables D9.3 and D2.3 we reported on a tool chain to transform access control policies written
in UWE (UML-modeling-language, cf. previous section) to XACML and from XACML to FACPL. XACML
stands for the OASIS standard eXtensible Access Markup Language (XACML) [65] and FACPL is an
abbreviation for Formal Access Control Policy Language, which emphasizes that, in contrast to XACML,
it is a formally founded language.

In this section we report on the toolchain extension implemented during the current reporting period.
This extended toolchain starts with the UWE to XACML transformation and includes a process to test the
exported policy [12].

Problem statement

In modern pervasive application domains, such as Service Oriented Architectures (SOAs) and Peer-to-
Peer (P2P) systems, security aspects are critical. Justified confidence in the security mechanisms that
are implemented for assuring proper data access is a key point. In the last years, XACML has become the
de facto standard for specifying policies for access control decisions in many application domains. Briefly,
a XACML policy defines the constraints and conditions that a subject needs to comply with for accessing
a resource and doing an action in a given environment. However, due to the complexity of the language,
XACML policy specification is a difficult and error-prone process that requires specific knowledge and a
high effort to be properly managed.

In recent years, model-driven approaches have been proposed for improving the definition of XACML
policies [65] to overcome intrinsic XACML language difficulties. Many methods are able to capture the
access control peculiarities by abstracting from the complexity of the language. However, a simplified
view could hide some security inaccuracies, due to an inappropriate use of model constructs. These
weaknesses can only be tackled by validating the final XACML policy. Model-driven proposals rarely

\(^3\)Acceleo. [http://www.eclipse.org/acceleo/](http://www.eclipse.org/acceleo/)
provide facilities for verifying the compliance of the derived policy with the requirements expressed in the model [54, 69].

In this deliverable, we make a step in this direction by presenting an integrated framework not only for designing security policies, but also for testing the compliance of the derived XACML sources with the initial models. Inspired by the conformance testing process we use some XACML-based testing strategies for generating appropriate test cases which are able to test functional aspects, constraints, permissions and prohibitions. The execution of these test cases, generated independently from the security model, provides (partial) input/output traces of the XACML policy execution. These data can be exploited for the construction of an additional model representing the XACML policy behavior, called the traces model. The compliance of the traces model with the associated security model is then assessed against some specified criterion.

Using a toolchain

The available proposals for verifying the consistency of access control policies with security requirements rely on the definition of specific properties of design (see for instance [54, 69]), which in some case could be very complex or difficult to express. The innovation of our proposal is based on the exploitation of a testing process for deriving the actual policy behavior and the use of typical model assessment techniques for discovering possible gaps between security and traces models.

Assisting users in design and testing of XACML policies is not possible without tool support. Therefore, we introduce a toolchain that includes the following components:

- Model-driven Policy Design: offers a tool for the graphical specification of security requirements and converts the model into a XACML policy;
- Test Case Generation and Execution: provides testing strategies to derive test cases and an engine for executing them on the XACML policy;
- Trace Analysis and Model Compliance: implements a methodology for analyzing the execution traces and for deriving the traces model. An oracle assesses the compliance of the traces model with the graphical security model.

Technical details about the “Model-driven Policy design” and “Test Case Generation and Execution” components are provided in the following. Currently we are finalizing the “Trace Analysis and Model Compliance” component of which we present the basic underlying ideas below.

Model-driven policy design

Modeling access control policies at a high level of abstraction has the advantage that policies are easy to understand and to maintain. UML-based Web Engineering (UWE) [25] is a notation for secure web applications which supports this approach. UWE uses the extension mechanisms provided by UML via the definition of a UML profile. UWE proposes to build different models for views such as content and navigation: (1) The content model, representing the domain concepts that are relevant for the web application and the relationships between them; (2) The role model, defining a hierarchy of user groups with the purpose of authorization and access control; (3) The basic rights model, expressing role based access control on the domain concepts; and (4) The navigation model, providing a graphical representation of the path the user can navigate in the web system. This model also includes security features as, e.g., authentication and secure connections.

For each view, UWE selects an appropriate type of UML diagram and provides a set of stereotypes, tag definitions, and constraints. For instance, the basic rights model is based on a UML class diagram which can be exported as XACML policy file. This file is used for the test case generation in the next step of our toolchain.

Test case generation and execution

X-CREATE (XaCml REquests derivAtion for TEsting) is a tool for systematic generation of XACML requests [14]. It takes into account the XACML policy structure, which basically consists of a Target and a set of Rules, specifying the access constraints and conditions. Specifically, X-CREATE implements four testing strategies that are based on a combinatorial approach of the values taken from the Target and Rules of a policy. In addition, random values are considered for negative
testing purposes. The first testing strategy (called Simple Combinatorial) uses values combinations for deriving simple requests containing one subject, one resource, one action and one environment. The main advantage of this strategy is that it is simple and achieves the coverage of the policy input domain represented by the policy values combinations. The second testing strategy (called XPT-based) exploits the XACML context schema for systematically deriving structurally more complex requests that are suitable when the Policy Decision Point (PDP) decision depends simultaneously on the values of more than one subject, resource, action and/or environment. The third strategy (called Incremental XPT) is an improvement of the XPT-based strategy which allows reducing the number of generated test cases. The last strategy (called Multiple Combinatorial) allows deriving requests having more than one subject, resource, action and environment.

**Trace analysis and model compliance** The Traces Creator derives the traces model by analyzing the XACML requests and the corresponding responses. In particular, for each request it maps: the subjects with the roles of the role model; the resources with concepts of the content model; and the actions with action stereotypes of the basic rights model. A positive response is translated into a stereotyped dependency between roles and concepts expressed in the request.

The Checker validates the compliance of the traces model against the original UWE model by considering domain specific assessment criteria. In particular, it analyzes whether the access permissions for each role are equally defined in both the traces model and the original basic rights model.

**Conclusions**

We defined a toolchain consisting of six tools. They can be used in an integrated way as well as separately. Used as a toolchain they allow for semi-automatically testing access control policies modeled with UWE. We are currently implementing the “trace analysis and model compliance” component. Further work is the evaluation of further case studies. Additionally, we are working on the refinement of the oracle and on a feedback loop for the generation of further requests.
2.2 Algorithmic verification

2.2.1 Sound security protocol transformations

It is well-known that security protocols are notoriously hard to get right. This motivates the use of formal methods for their design and development. The last decade has witnessed substantial progress in the formal verification of security protocols’ properties such as secrecy and authentication. However, methods for transforming protocols have received much less attention.

Protocol transformations are interesting for at least two applications: we can use them (1) to develop (families of) protocols by refinement [18,30,31,41,67,80] and (2) to abstract existing protocols for the more efficient tool-based verification of their properties [44]. Abstraction and refinement correspond bottom-up and top-down views on (the same) protocol transformations. To be useful, protocol transformations must be sound with respect to a relevant class of security properties, i.e., refinement must be property-preserving, or equivalently, abstraction must be attack-preserving.

In this work, we propose a class of syntactic protocol transformations covering a wide range of protocols and security properties. Following Hui and Lowe [44], we support both message-based transformations, which we lift to protocol roles, and structural transformations, which directly operate on protocol roles.

Message-based transformations include the removal of hashes or encryptions, pulling cleartext fields out of an encryption, and rearranging pair components. To guarantee the uniform transformation (e.g., removal) of variables and the messages they are supposed to receive, we work with typed messages. We use the type system of Arapienis and Duflot [4], which enables a fine-grained control over the message transformations. We establish the soundness of our typed transformations with respect to an expressive property specification language based on [57].

Motivating example

We discuss the abstraction and refinement of key establishment protocols. We first take the abstraction view and defer a brief discussion of the refinement view to the end of this section. We start from a core version of Kerberos IV, called K4, which we simplify in several steps with the aim of optimizing the performance of verification tools. In Alice&Bob notation, the protocol K4 reads as follows.

\[
\begin{align*}
K4(1). & \quad A \to S : A, B, n_A \\
K4(2). & \quad S \to A : \{B, t_S, n_A, k_{AB}, \{A, t_S, k_{AB}\}_\text{sh}(B,S)\}_\text{sh}(A,S) \\
K4(3). & \quad A \to B : \{A, t_S, k_{AB}\}_\text{sh}(B,S), \{c, t_A\}_k_{AB} \\
K4(4). & \quad B \to A : \{t_A\}_k_{AB}
\end{align*}
\]

The security properties we are interested in include: (P1) the secrecy of the session key \(k_{AB}\), (P2) \(A\) authenticates \(S\) on \(k_{AB}, n_A\), and \(t_S\), and (P3) \(A\) and \(B\) authenticate each other on \(k_{AB}\) and \(t_A\). To improve the performance of verification tools, we remove protocol elements that we deem unnecessary for a given property to hold and verify that property on the simplified protocol. If there is no attack then the soundness of our abstractions allows us to conclude that the original protocol also satisfies the property.

In the first abstraction step, we pull \(B\)’s ticket out of the encryption in message K4(2). The result is the core of Kerberos V, called K5, which differs from K4 as follows.

\[
\begin{align*}
K5(2). & \quad S \to A : \{B, t_S, n_A, k_{AB}\}_\text{sh}(A,S), \{A, t_S, k_{AB}\}_\text{sh}(B,S)
\end{align*}
\]

In the second step, we eliminate the forwarding of \(B\)’s ticket by \(A\) by applying structural transformations. This yields protocol K3, on which we verify mutual authentication of \(A\) and \(B\) (P3). We omit the message K3(1) which equals K5(1).

\[
\begin{align*}
K3(2). & \quad S \to A : \{B, t_S, n_A, k_{AB}\}_\text{sh}(A,S) \\
K3(3). & \quad S \to B : \{A, t_S, k_{AB}\}_\text{sh}(B,S) \\
K3(4). & \quad A \to B : \{c, t_A\}_k_{AB} \\
K3(5). & \quad B \to A : \{t_A\}_k_{AB}
\end{align*}
\]

In the third step, we remove the key confirmation phase, i.e., messages K3(4) and K3(5). For the resulting protocol, K2, which we omit here, we verify the authentication property (P2).
In a final transformation, we remove the server timestamp \( t_S \) and the initiator nonce \( n_A \). The result is protocol \( K_1 \) for which we verify secrecy (P1).

\[
K_1(1). \quad A \to S : A, B \\
K_1(2). \quad S \to A : \{B, k_{AB}\}_{sh(A,S)} \\
K_1(3). \quad S \to B : \{A, k_{AB}\}_{sh(B,S)}
\]

We can also view these transformations in the other direction, as a development of \( K_4 \) by refinement. We start from the abstract protocol \( K_1 \) satisfying session key secrecy (P1) and add new properties or modify the protocol structure with each refinement step. We verify properties (P2) and (P3) for \( K_2 \) and \( K_3 \), respectively, knowing that they are preserved by further refinements. By refining given protocols in different ways, we can develop entire protocol families, whose members share structure and properties. For example, most server-based key establishment protocols can be derived from \( K_1 \).

Contributions

We make the following contributions. First, our work provides a sound formal underpinning for protocol transformations, which can serve as a foundation for rigorous security protocol development by refinement as well as for the abstraction of existing protocols. As an example of the latter, our approach helps to improve the performance of security protocol verifiers that are sensitive to message sizes such as SATMC [6]. Second, we extend existing work with respect to the expressiveness of the protocol specifications, the protocol transformations, and the preserved properties. In particular, we extend [44] in several ways: (1) we clarify and formally justify the application of transformations to protocol specifications, which contain variables not only ground terms as in [44]; (2) we support composed keys under a mild restriction; (3) we cover additional transformations (e.g., splitting encryptions) including many of those in [26, 30, 31]; and (4) we extend soundness to a more expressive property language based on predicates expressing event occurrence and ordering, intruder knowledge, and including quantification over thread identifiers.

More detailed information about this work is given in [63]. The full version of that paper [62] includes the proofs of all our results and the treatment of structural transformations.

2.3 Cryptography

2.3.1 Quantum-computing-resistant cryptographic algorithms

The design of secure security protocols is still a difficult task, even though formal methods are increasingly used and refined to develop such protocols and to provide proofs of their security. In all these attempts it is implicitly assumed that the underlying cryptographic mechanisms are secure.

However, it is a common that all practically usable cryptographic algorithms have only a restricted lifetime. Consequently, obsolete algorithms have either to be replaced by new ones (e.g., DES by AES) or – this happens mainly in the case of asymmetric algorithms – the lengths of parameters and keys have to be enlarged.

Completely different from this common situation is the potential impact that quantum computing and the availability of a quantum computer would have on cryptography in general. The purpose of this work is to describe the potential consequences of the availability of a quantum computer for cryptography and to give – from a today’s point of view – an overview on cryptographic algorithms resistant against quantum computing.

The algorithms of Grover and Shor and their impact

The standard reference for quantum computing is still the famous book of Nielsen and Chuang [64]. The relevance of quantum computing for cryptography in general results from the following two algorithms:

**Grover’s algorithm:** Grover’s algorithm [40] is a probabilistic quantum algorithm for searching for a specific element in an unsorted database with \( N \) entries. Its time complexity is \( O(\sqrt{N}) \) and the space requirements are of size \( O(\log N) \). Roughly speaking, Grover’s search algorithm halves the effective key
size of all symmetric cryptographic algorithms. Its impact on symmetric cryptography is moderate and easy to manage. An effective countermeasure against Grover’s algorithm is just to double the key size. The Luby-Rackoff construction [50] provides a general framework for this approach. Symmetric algorithms like AES with key length of 256 bits are immune against Grover’s search algorithm.

**Shor’s algorithm:** Shor’s algorithm [78] in its generalized form is a probabilistic quantum algorithm to find a hidden subgroup $H$ in a finite abelian group $G$, i.e., to find a system of generators of $H$ and the order $|H|$. If the order of $G$ is of size $2^n$ then Shor’s algorithm runs in time $O(n^3)$ using a quantum register of length $3n$. As soon as a quantum computer working with quantum registers of this length will be available, we have to face dramatic consequences for the currently used asymmetric cryptographic schemes: All the schemes based on the integer factorization problem (RSA, Rabin) and on the discrete logarithm problem in a finite abelian group (e.g., Diffie-Hellman, ElGamal, DSA, and ECDSA) will collapse immediately and need replacement.

**The current asymmetric post-quantum candidate algorithms**

Asymmetric cryptographic algorithms that are considered immune against quantum computing attacks are now generally called post-quantum algorithms. A good overview over the currently discussed asymmetric cryptographic schemes that are with today’s knowledge post-quantum in this sense is given in the book edited by Bernstein, Buchmann, and Dahmen [11]. As in the case of today’s asymmetric cryptography these schemes are based on certain mathematical problems that are considered hard to solve.

The most prominent of these problems and the adjacent post-quantum systems are the following:

- **Code based systems:** The underlying problem is the decoding problem of a general linear code. The most prominent of this schemes is the McEliece public-key encryption scheme.

- **Multivariate Polynomial Systems:** The underlying problem of these schemes is to solve a system of multivariate polynomial equations over a finite field. Specific cryptographic schemes are given by Patarin and Matsumoto and Imai, et al.

- **Lattice based schemes:** The typical underlying problem is to find the shortest vector in a lattice. The most important example based on this class of problems is the NTRU scheme.

- **Merkle-like schemes:** These schemes are based on the Lamport-Diffie concept of one-time signatures (OTS). To make the impractical OTS usable the Merkle approach combines OTS with a set of keys arranged in a binary tree.

In our work, we describe the above mentioned schemes and analyze them with respect to performance and resource requirements. Compared with the asymmetric cryptographic algorithms that are in use today all the above mentioned schemes have some specific drawbacks, e.g.:

- Code based systems work with extremely large keys (in the order of MBytes). Up to now there exists no really convincing concept for digital signatures available based on this approach.

- The most practical lattice based schemes (NTRU-family) are covered by patents. But, more importantly, there are severe security concerns related to NTRU-signatures.

- The Merkle-like schemes allow only a pre-defined number of digital signatures. This number may be arbitrarily large, but this choice influences the time necessary to verify the authenticity of the used public key.

For more details we refer to the literature or, again, to our forthcoming report.
Conclusions

The future development in the field of post-quantum algorithms seems to be completely open. If a quantum computer implementation of Shor’s algorithm would be available today one would probably replace the currently used asymmetric encryption and key exchange schemes schemes by the the McEliece scheme or by the NTRU encryption scheme. The most promising alternative to the today’s signature schemes – despite all the restrictions related to the number of possible signatures – might be based on the Merkle-approach. The security assumptions for Merkle-like signatures are minimal, one needs only a secure one-way or hash function, an example of such a scheme is XMSS [23].

However, as time goes on, the development of post-quantum cryptography might even proceed in a completely different direction. The fact that the currently preferred post-quantum algorithms are not vulnerable to Shor’s algorithm is no guarantee that these schemes cannot be successfully attacked in a different way - either classically or by future quantum algorithms. Furthermore, there are other areas of research that might yield more efficient post-quantum schemes. One idea is based for instance on the concept of generalized discrete logarithms in finite non-abelian groups, see [48].

Just recently we started to investigate the question of integrability of the currently discussed post-quantum schemes into well-known security protocols (e.g., IPSec, TLS, and IKE). This work is ongoing. We plan to publish the results of this investigation in a separate document.
3 Assurance in Implementation

This section focuses on assurance techniques that ensure the security at the implementation level. Internet application security can be validated through testing. Here the focus is on the automatic generation of effective test cases. Testing is covered in Section 3.1. Runtime monitoring and enforcement is a complementary technique to ensure that the running system satisfies the required security properties. This is the topic of Section 3.2.

3.1 Testing

3.1.1 Mutation-based testing of security protocols

Bridging the gap between an abstract attack scenario derived from a specification and a penetration test on real implementations of a protocol is still a non-trivial issue. Therefore we propose an architecture for automatically generating abstract attacks and converting them to concrete tests on protocol implementations. In particular, the objective is to improve previously proposed black-box testing methods in order to semi-automatically discover new attacks and vulnerabilities. A related approach to test concretization has been proposed independently [7]. The authors proposed an ad-hoc procedure to compile their attack scenarios. The present approach directly re-uses a previously-existing procedure that was designed for the Avantssar platform [5]. It is also important to notice that the work in [7] does not leverage mutation techniques [28] as it is done here. The approach below initiated in [28], then applied in [39] is detailed in a journal submission.

Platform architecture

Given a security protocol specified in high-level specification language (e.g., HLPSL), the attack trace produced by a model-checker (possibly following mutation as explained in Section 3.1.1 of Deliverable D9.3 [83]) is rather abstract. In order to be able to detect real attacks that affect protocol implementations, it is mandatory to provide a platform that performs both message format conversion, from a formal level to the implementation level, and real communications with the System Under Test. We now describe this platform’s architecture in terms of its components with their functionalities and interactions. It displays three main components, each with a specific role:

1. **Attack Trace Compiler**: identifies agents, messages types, and elementary operations.
2. **Scenario Execution Engine**: generates (resp. retrieves) outgoing (resp. incoming) messages.
3. **Attack Simulator**: simulates the scenario on real communication channels.

As shown in Figure 3.1, the testing environment takes as inputs the attack trace and the mutated model of the considered protocol, and returns an indication whether the considered attack on the considered implementation exists or not. The **Attack Trace Compiler** collects intruder initial knowledge from the protocol specified in high-level language (e.g. HLPSL) and follows the attack trace instructions to build a detailed attack scenario. The latter describes unambiguously the actions that should be performed by the intruder when executing the attack. For each step $I ightarrow B : M$ in the attack trace, where $I$ is an agent controlled by the intruder, one needs to check whether the intruder can compose $M$ at this step.

The **Scenario Execution Handler** is responsible of translating the attack scenario, from the formal level to the implementation level. Since the execution environment is designed for the implementation level, the exchanged messages are real network messages. Therefore, it is necessary to map these terms to concrete messages stored in the **Data Store** module. Operation execution is held with the functionality provided by the **Primitive Holder**.

After mapping a formal message to the real format, the **Scenario Execution Handler** processes emission and reception operations. In these cases, it sends a request to the **Attack Simulator** module, which is the interface of the platform with the external environment. At the formal level, the protocol model abstract some fields existing in a real implementation, which need to be restored at the concrete level. The **Attack Simulator** is responsible for the conformance of the exchanged data with the protocol model, meaning that...
it has to identify the relevant fields and retrieve data from the System Under Test (SUT) response (case of receiving operation) and to instantiate the relevant fields in the request message (case of sending operation). In general, the **Attack Simulator** tasks include: (i) creating the real communication channels, (ii) sending messages, and (iii) receiving messages.

This module is also responsible for validating the execution of the whole attack scenario. By monitoring the SUT states, it can easily detect states that correspond to a successful attack.

**Payment protocol application**

As a proof-of-concept, the whole process was targeted to **PayPal Standard** protocol implementations, in a recent version of the **Magento** platform [51].

In the considered case study the well-known Paypal server is used as an example of **Cashier-as-a-Service (CaaS)** server. This third-party integration introduces a complexity in the payment protocol implementation within the e-commerce application which brings new security issues. A dishonest shopper can make Web-API calls of methods existing on the e-commerce application with well-chosen arguments and in an arbitrary order so that he can shop products for free or alter the way the payment is verified. This shows that to have communications over **HTTPS** does not prevent severe attacks against e-commerce applications.

The checkout process begins when the “Pay Now” button on the merchant web site is clicked. This operation directs the shopper’s browser to the PayPal website where he is invited to give his PayPal buyer account some credentials to continue the purchase process. If the information entered by the user is correct, the shopper is again redirected to a payment success merchant Web page. Behind the scene, there are **HTTPS** interactions between the three parties, who communicate by calling Web-APIs exposed by the merchant and the CaaS. Such APIs are essentially dynamic web pages and are invoked through **HTTPS** requests. A client sends an **HTTPS** request through an **URL** with a list of arguments and receives an **HTTPS** response, often a Web page dynamically constructed by the server, as the outcome of the call.

The formal model of the protocol was designed using the PayPal documentation [68] and some traffic analysis. To examine the traffic, Magento was deployed on a local **xampp** server [89] and **HTTPS** traffic capturing tools were used, such as Fiddler [38], retrieving all the **HTTPS** exchanged messages between...
the involved entities during a checkout process. Figure 3.2 depicts all the exchanges of HTTPS messages.

First, the client starts with sending his identity and the order description informations: product details, shipping address, billing address, etc. The merchant redirects the client to the authentication page on the Paypal server. The merchant's response also contains the product purchase amount signed by his private key. Then, the client sends his account credentials to the Paypal server with the received purchase amount (still signed). Note that a nonce is used to identify the session initiated by the client. Finally, the Paypal server validates the received information and returns a Success payment page.

The use of the mutation operators has enabled the removal of the signature on the product amount, which is normally performed by the merchant in order to guarantee the integrity of the data sent to PayPal. After applying the mutation process, the model checking tool Cl-AtSe has been used to verify the protocol and to generate an abstract attack trace. The attack was concretized and has been replayed entirely in the environment in which the experimentation was conducted, meaning that this implementation of Magento contains a logical security flaw that consists in the absence of a signature of the purchase amount. To exploit this flaw, a malicious attacker can modify the amount to pay, and purchase expensive items by paying a ridiculous price. This logical flaw was discovered recently by the NBS Systems company.

Conclusions

As a proof of concept of our semi-automatic test concretization method, we have been able to rediscover a real attack on an e-commerce application. An overview of the approach can be found in [39]. In future works we plan to refine the test verdict analysis in order to decrease the number of false positive alerts.

3.1.2 VERA: A flexible model-based vulnerability testing tool

Usually, in order to state what is meant by the security of a system, one defines high-level security requirements, determined by the assets that one wants to protect. The goals are integrity, confidentiality, and availability of the resources associated with those assets. The requirements are enforced via mechanisms that provide authentication, authorization, secure channels, etc. As a consequence, secondary goals appear, for instance only authorized people have access to a certain interface, a directory, or to a certain cookie. Goals break down into mechanisms plus new more lower-level requirements. At the end one may have a goal of the form: the system must not have cross-site scripting (XSS) vulnerabilities (shorthand for: the system should not react in a particular way to a particular input), because violating this goal might result in a violation of the higher level security requirements (such as confidentiality or integrity breaches).

In many cases it is not necessary to find a full-blown attack, but it is enough to find low-level vulnerabilities. This is among others justified by the cost/benefit relation of vulnerability testing: usually the presence of a vulnerability is a pre-requisite to deploy an attack, but to actually exploit a vulnerability is a time-consuming task (if at all possible). Consider the following example: A tester finds that when giving some inputs to a web application, a malformed SQL statement error is reported. This could mean that there is a way to craft the input with malicious SQL injections, but to find the exact form in which these injections are successful is in general not trivial, in particular if the source code is unknown (as in black-box testing). Conservatively fixing the application to eliminate the unwanted behavior (SQL error) offers a reasonable compromise between security and the testing effort.

Thus, security testing often concentrates on vulnerability testing. As of today, there exist an abundant number of tools for aiding developers and penetration testers to spot common software security vulnerabilities. However, testers are often confronted with situations where existing tools are of little help because of the following reasons:

- The existing tools do not account for a particular configuration of the system under test (SUT).
- The existing tools do not include tests for certain vulnerabilities. For instance, the SUT could use a particular authentication mechanism (such as a proprietary protocol) and a recent/rare database version. It is likely that most available tools will not cover the authentication mechanism and might not have information about known vulnerabilities of the database model used by the SUT.

In this situations the tester would benefit from extension mechanisms to the available tools. Some tools (like the non-free version of BURP W3AF) allow to write such extension plug-ins. These plug-ins must be written in the programming language of the tool and imply the learning curve of the tool's API. The alternative consists in directly writing scripts for the task in hand.

We have proposed a tool named VERA, standing for “VERA Executes the Right Attacks”, which allows testers to define attacker models by means of extended finite state machines. In this way, testers can define new tests where the payloads and the behavior are cleanly separated and that abstract away from low-level implementation details such as HTTP requests. In the following, we briefly explain the architecture of the VERA tool.

The VERA tool

The VERA tool is an extensible framework based on the concept of extended finite state machines that allows the creation and execution of attacker models targeting generic vulnerabilities of Web applications in a black-box fashion. The attacker models for VERA can be collected in libraries targeting specific vulnerability types across multiple types of web applications.

The basic functionality of an attack is encapsulated within the attacker models. An actual attack however often needs additional data that is ill suited to automata. An example for this would be a list of passwords in a brute force attack. We have made a design choice to outsource this type of information, which is not application specific, into standardized instantiation libraries. These libraries can be accessed by the attacker models as arrays, in essence instantiating the attacks with specific values. VERA creates an instance of the attacker model, and runs it against the SUT using the values from the instantiation libraries. The tool may also consult configuration files which contain system specific information about the SUT, e.g., the URL address of the Web server under test. If needed, the security expert may guide the VERA tool manually.

In the following we illustrate the VERA's attacker models with the help of an example. Attacker models can be seen as an extension of Mealy machines [56] with guarded transitions and variables. One can also see similarities between the attacker models and UML statecharts. However, in contrast to UML statecharts, attacker models do not have a notion of composition.

Let us consider the example of Figure 3.3. In the first transition the attacker starts the interaction with the system by sending a message requesting a particular URL. Subsequently, a message is received by the attacker that does not trigger an immediate reaction, but only a change in the configuration. With the new configuration (the new values of \( x \), \( i \) and \( j \)) we have two possible transitions: i) to the final state and ii) the to the third state.

The transition to the final state is triggered when there are no more fields to check (or the page pointed by URL doesn't contain any fields), and the attacker gives up by outputting failure. The other
two transitions, to and from the third state, represent a configuration change for the variables $j$, $l'$ (for the outgoing edge), and $i$ (for the incoming edge). In the lower-right state the attacker sends a message with the $j$-th payload in IO (to be delivered to the $i$-th field in the page) without receiving a message.

Subsequently the attacker receives a message and changes his state (thus he reacts) according to its content. If the message does not contain the expected output the state will change in order to check the next payload or the next field; on the other hand if the attacker receives the expected output he has found a vulnerability thus outputting success.

**Conclusions**

We have used the VERA tool to encode a number of practically relevant attacker models for Web applications, including models for SQL injection, XML wrapping, and cross-site request forgery. We have so far explored primarily Web vulnerabilities; VERA is however sufficiently flexible to be applied in other contexts as well. Currently, within Siemens and SAP we are evaluating the benefits of our approach. A thorough report on our experiments in this direction will be subject of future work. For further details about the present work, we refer the interested reader to our conference paper [19].

**3.1.3 Test case prioritization for XACML policies**

Among security mechanisms, one of the most important components is the access control system, which mediates all requests of access to protected data and ensures that only the intended (i.e., authorized) people can access them, and that these intended users are only given the level of access required to accomplish their tasks.

It is hence evident that the trustworthiness of a security framework is strongly dependent on the proper management of accesses authorization, and the latter mechanism in turn relies on the correct specification and implementation of security policies. To guarantee that policy implementation conforms to the specified policy, it is important to perform verification of the policy in addition to validation that could be done through testing. This section focuses on the latter by tackling the issue of testing XACML based policies.

In deliverable D9.1 and D9.2 several approaches have been overviewed and proposed to generate XACML tests. A common limitation of these tools is that they produce a huge number of requests. Due to budget and time limitations, it is impossible to run all those tests and check that the result is correct (this step can only be done manually). It is therefore important to select the best subset to run instead of running all the tests. This section focuses on solving this specific issue, namely, XACML test prioritization.

Test prioritization has been widely investigated in the field of software testing, and an approach that is currently considered very promising is based on the notion of test similarity: the intuition behind similarity-based prioritization is that if only a limited subset of test cases within a large test suite can be executed, then it is convenient to pick those that are the most dissimilar according to a predefined distance function. Thus test case prioritization [37] aims at defining a test execution order according to some criteria (i.e,
coverage) so that tests which have a higher priority are executed before the ones that have lower priority. In [72], several test prioritization techniques have been described and have been used to augment fault detection capabilities of test suites. There has been a lot of research in test case prioritization in the last years, however in the context of XACML Policies, we considered the vision of [34]. In their work, based on the hypothesis that mutation faults can be representative of real faults, the authors have shown that prioritization techniques are effective to improve fault detection rate.

We therefore propose to adapt similarity-based prioritization to reduce XACML test cases. To do this, we need to capture and specify what is a suitable notion of distance between XACML requests. To the best of our knowledge, our approach is the first attempt to introduce a prioritization strategy in XACML policy testing.

To evaluate the prioritization strategies, we consider the fault detection criteria that enables us to evaluate the quality of the ordered tests. We rely on mutation analysis to inject faults into the policy and challenge the tests to detect these seeded faults. The goal is to end up with XACML tests ordered in a way that enables reaching quickly a high mutation score. Indeed, by using the tool XACMUT [15] we developed, mutation analysis has been applied on access control policies to qualify security tests. By means of mutation operators, the policy under test is modified to derive a set of faulty policies (mutants) each containing a fault. A mutant policy is killed if the response of a XACML request executed on the mutant policy differs from the response of the same request executed on the original policy.

Thus we contribute by the the definition of similarity metrics based on XACML policies for comparing XACML requests and the introduction of the first prioritization technique for XACLM policy testing. Further details are provided in the journal submission [13].

**Similarity metrics**

Similarity is an heuristic which is used here to order access control requests (i.e. the test cases). Previous work on model-based testing, such as [42] have shown that dissimilar test suites bestow a higher fault detection power than similar ones. Experimental results obtained show that two dissimilar access control requests find more access control faults than two similar ones. In particular considering a test suite of \( n \) access control requests \( \{R_1, ..., R_n\} \) the approach proposed considers two steps: The first step involves the definition of a distance metric \( d \) between any two access control requests \( R_i \) and \( R_j \), where \( 1 \leq i, j \leq n \). This metric is used to evaluate the degree of similarity between two given requests: the highest the resulting distance, the most different the two requests. The second step is the ordering of these \( n \) requests. To this end, we first compute the distance between each pair of requests. Then, a selection algorithm uses the distances to select the most dissimilar requests, resulting in a list where the first requests selected are the most dissimilar ones.

In our experiment we considered two possible methods for calculating distance metric \( d \) between any two access control requests. The former, called *simple similarity*, is mainly based on lexical distance of the requests parameters (subject, action, etc.). In this case the distance \( d_{ss} \) can be generally defined as follows:

\[
d_{ss} : R \times R \rightarrow \mathbb{R}_+ (R_i, R_j) \mapsto d_{ss}(R_i, R_j).
\]

The latter, called *XACML similarity*, takes in consideration not only the lexical distance, but also the applicability of the requests to the policy. In particular the policy can be used to evaluate which of the two requests better satisfy the policy rules and the targets (policies and policy sets targets).

In this case, the distances between requests are policy-dependent. The distance \( d_{xs} \) can be generally defined as follows:

\[
d_{xs} : R \times R \times P \rightarrow \mathbb{R}_+ (R_i, R_j, P) \mapsto d_{xs}(R_i, R_j, P).
\]

Thus the *XACML similarity* considers the policy and computes an applicability vector. The idea is to go through the policy at all its levels from the policy set target to the rules targets and compare the request target to each level target. If the request target matches that level target (policy set, policy or rule) then it is considered to be applicable and this information is stored in the applicability vector. In addition, we consider each rule parameter (subject, action, resource and environment) and compare it to the target corresponding parameter. Here as well, if that parameter matches the related target parameter, then it
is considered to be applicable. Finally, we end up with an applicability vector for each request. Then the distance is computed comparing each two request vector.

In both cases, simple similarity and XACML similarity, the main idea is to define a distance metric that uses available information (i.e., parameters of the requests and the policy structure) to evaluate the degree in which two requests can be considered similar or dissimilar. In [13] we adopt the convention that the higher the resulting distance value is, the most dissimilar the two requests are; a distance value equal to 0 means that two requests are identical.

In practice the proposed algorithm aims at selecting the request that is the most distant to all the requests already selected during the previous steps of the approach. To this end, we first select the two XACML requests sharing the highest distance. These two requests are then removed from the set of XACML requests to prioritize. Then the request sharing the maximum distance from all the requests already selected are chosen. For each remaining request we sum the individual distances with the distance of the request. Then, the maximum is obtained by comparing these set values. The selected request is removed from the original set and this process is repeated until all the request have been selected.

### Ordering the access control requests

This section presents the prioritization of the requests using the distances defined in the previous sections. The idea is to order the requests so that we will execute first the most dissimilar ones, i.e., those sharing the higher distance. In this work, the technique proposed in [43] is adapted to prioritize the XACML requests. Algorithm 1 details the approach.

**Algorithm 1 Prioritization**

1: **input**: $S = \{R_1, ..., R_n\}$  
2: **output**: $L$  
3: $L \leftarrow []$  
4: Select $R_i, R_j$ where $\max (d(R_i, R_j)), 1 \leq i, j \leq n$  
5: $L.add(R_i)$  
6: $L.add(R_j)$  
7: $S \leftarrow S \setminus \{R_i, R_j\}$  
8: while $\#S > 0$ do  
9:     $s \leftarrow \text{size}(L)$  
10:    Select $P_i \in S$ where $\max (\sum_{j=1}^{s} d(R_i, L.get(j))), 1 \leq i \leq n$  
11:    $L.add(R_i)$  
12:    $S \leftarrow S \setminus \{R_i\}$  
13:   end while  
14: return $L$

Informally, the algorithm aims at selecting the request that is the most distant to all the requests already selected during the previous steps of the approach. To this end, we first select the two XACML requests sharing the highest distance (Alg. 1, line 4). These two requests are then removed from the set of XACML requests to prioritize, $S$ (Alg. 1, line 8). The next step consists in selecting the request sharing the maximum distance to all the requests already select. For each remaining request in $S$, we sum the individual distances with the request. Then, the maximum is obtained by comparing these set values (Alg. 1, line 11). The selected request is removed (Alg. 1, line 14) and this process is repeated until all the request have been selected.

### Policies and experimental setup

In [13] we applied the proposed similarity-based approaches on 6 policies detailed on Table 3.1. There some information regarding the sizes of the policies and the number of subjects, actions, resources and environments for the six policies used in our experiments are provided and compared the results against the random prioritization and the sub-optimal prioritization (computed based on the mutation coverage). We also compare with the default request order (considered as a default prioritization) provided by the tool X-CREATE [16].

From the obtained results, we inferred the following conclusions:
Table 3.1: Description of the six policies

<table>
<thead>
<tr>
<th>Policy Name</th>
<th>Rules</th>
<th>Subjects</th>
<th>Actions</th>
<th>Resources</th>
<th>Environments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASMS</td>
<td>117</td>
<td>8</td>
<td>11</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>itrust</td>
<td>64</td>
<td>7</td>
<td>9</td>
<td>46</td>
<td>0</td>
</tr>
<tr>
<td>VMS</td>
<td>106</td>
<td>7</td>
<td>15</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>continue-a</td>
<td>298</td>
<td>16</td>
<td>4</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>LMS</td>
<td>42</td>
<td>8</td>
<td>10</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>pluto</td>
<td>21</td>
<td>4</td>
<td>1</td>
<td>90</td>
<td>0</td>
</tr>
</tbody>
</table>

• Effectiveness of the XACML-Similarity approach: We noticed that for all policies the XACML-Similarity provides always better results and is close to the nearly optimal solution (the greedy results). This indicates that taking into considering the policy is very useful when it comes to prioritizing.

• Lack of effectiveness of Simple-Similarity: The obtained results show that the simple similarity is providing poor prioritization results. Ignoring the policy and replying only on the requests content to perform similarity prioritization leads obviously to poor results that are similar to random prioritization.

• X-CREATE results are similar to random prioritization results: Interestingly, the six policies results demonstrated that the default order in which X-CREATE tool creates the requests is providing mutation-killing capability similar to the random one. This result is an important factor in favor of the need to apply prioritization techniques (like similarity) because the testing strategy generation order of the requests may be not the best one.

Further details about this experiment are provided in [13]

Conclusions

In this section, we presented a new approach based on similarity aiming at prioritizing tests in the context of XACML access control systems. We proposed two similarity-based prioritization metrics: the first strategy is the simple similarity, which is policy-independent and involves comparing the content of requests; the second approach is called XACML similarity and considers the applicability of the requests to the XACML policy. We performed an empirical study to evaluate the effectiveness of the simple and the XACML similarity metrics applied to the test suites related to a set of six real world XACML policies. The results showed that the second approach is effective and provides a mutation coverage that is significantly better than random prioritization and close to a sub-optimal heuristic cognizant of the requests effectiveness.

3.1.4 Testing of PolPA-based usage control systems

Authorization systems enable the specification of security policies that rule various protection aspects such as: the level of confidentiality of data, the procedures for managing data and resources, the classification of resources into category sets with different security requirements.

Among the several existing systems, in this deliverable we focus on the PolPA authorization system, which has been defined in [52]. PolPA exploits a process-algebra-based security policy language that supports history-based control and Usage Control (UCON). Hence, the PolPA authorization system can control the sequence of security-relevant actions performed by the user so as to prevent the execution of an action when a policy is not satisfied. Moreover, it allows policy makers to express conditions that control the usage of resources, i.e., these conditions are continuously evaluated all along the access time, so that an access right can be revoked as soon as it is violated.

From an architectural point of view, the PolPA authorization system includes several components, such as: the Policy Enforcement Point (PEP), which enforces the policy decisions, the Policy Decision Point (PDP), which performs the decision process to determine whether (according to the defined security...
policies and to the current values of the attributes of the users, resources and environment) an access should be granted or denied, the Policy Information Point (PIP), which is in charge of retrieving the attributes of users, resources, and environment, and the Policy Administration Point (PAP), which supports the editing and the storing of the different policies.

All the above components are critical from a security point of view and would require careful verification and testing. In this deliverable, we refine and expand our preliminary proposal for a PolPA testing framework presented in the Deliverable D9.3 [83], where the tested PDP only supported history-based security policies. Here, instead, we propose the complete process covering both history-based and continuous policy enforcement testing.

Available solutions for testing PDPs are usually focused on standard access control models and more popular access control languages, such as OrBAC, RBAC, or XACML, and cannot be easily transferred into the UCON environment. To the best of our knowledge, there are no suitable testing technologies for validating the PDP implementation of continuous control of the user accesses. Hence, we propose a testing framework customized for the PolPA language, specifically designed to deal with history-based UCON security policies. Thus, we extend the already proposed framework by including a test generator for continuous policy enforcement specifically conceived for addressing the dynamic runtime behavior of the PDP required for prompt access revocation and an automated oracle able to determine whether a test has passed or failed for the continuous policy evaluation. A more complete description of the proposed framework is in [17].

Motivations

In order to test the continuous policy enforcement functionality of the PDP, we need to focus on the mutability of attributes, and specifically on the ability of the PDP to promptly react to any attribute change during the execution of an access. The PIP interacts with the PDP for managing the attribute values. Notice that the PDP can provide different replies to distinct evaluations of the same authorization request, just depending on the values of the attributes, which dynamically change over time. The influence of the external environment, expressed in terms of mutability of attributes, therefore plays a key role in the continuous policy enforcement. The PDP misinterpretation of these attributes’ changes could have critical consequences on the overall policy enforcement and on the system security. Possible erroneous situations could be:

- Before the evaluation of an authorization request, the PDP asks the PIP for the updated values of the attributes required to perform the decision process. However, an error occurs in the attribute value exchange phase, so the PDP does not authorize the access to the resource. For instance, the PDP used a corrupted attribute name in the attribute value request, or it performed an error extracting the new attribute value from the response received from the PIP.

- During the execution of an authorized access, the PIP notifies the PDP of the change of some attributes that, according to the policy, should cause the revocation of this access. However, the PDP does not manage the PIP message correctly, so the access is not interrupted and the user keeps on utilizing the resource although the corresponding right is not valid any more. For instance, the PDP performed an error extracting the new attribute value from the message received from the PIP, or did not keep trace of the ongoing accesses correctly.

Both of the above examples are extremely critical from a security point of view so the occurrence of these situations should definitely be prevented. Moreover, the examples show that only the prompt acquisition of the updated values of the attributes can assure correct enforcement of the security policy by the PDP.

Testing strategy

The malfunctioning of the PDP is mainly related to faults in evaluating predicates that involve corrupted attribute values, we propose a testing strategy based on predicate coverage criteria. The strategy derives a set of test cases suitable for covering all cases in which the attributes values could be modified during the access life cycle, i.e., from the access request to the access termination. Specifically, a test case stimulates the PIP to change the value of one (or even more) attribute(s) so as to enforce the PIP and PDP to interact for exchanging the new attribute value, re-evaluating the PolPA policy predicates, and possibly
enforcing the revocation of one (or more) access(es). In PoIPA, a predicate is a boolean expression with one or more logically connected conditions, where each condition is, in turn, a boolean expression involving the values of mutable attributes related to the user, to the resource, or to the environment.

Thus, the proposed test strategy combines the standard technique for conditions coverage with a methodology for simulating all the possible ways in which attributes can change during the access life cycle. Therefore the same predicate can have different PDP evaluations depending on when PIP component notifies the PDP of the attributes change. The analysis of PoIPA policy execution allows us to define exactly the moments in which a new attribute value can be sent by the PIP to the PDP during the evaluation of an access request. For the sake of simplicity, avoiding the complex details about the exact access request execution, these moments are:

- **before the access request evaluation**: The attribute value changes just before the evaluation of the access request against the security policy. In this case, the new value is exploited to evaluate such a request. If the requested access is authorized, but the evaluation of a predicate associated to a revocation command returns true, the PDP must revoke the previously authorized access.

- **during the access execution**: The access request is evaluated against the security policy and the access is granted, and the PDP subscribes to receive attribute updates from the PIP. If, during the execution of the access, the PIP notifies the PDP of the attribute changes and the evaluation of the predicate associated to a revocation command with the new value results to be true, the PDP must revoke the user access.

- **after the access execution**: The access request is evaluated against the security policy and the access is granted. The access is executed and terminates normally. Only after the access termination the PIP notifies the PDP of the attribute changes. In this case, independently from the evaluation of the predicate associated to a revocation command, the PDP goes ahead with the evaluation of the next request.

In particular, the proposed testing strategy is based on the Multiple Condition Coverage (MCC) approach [55]. A multiple condition is made up from one or more conditions, which are combined by logical operators (and, or, not). To get 100% coverage, for each multiple condition, all combinations of true and false for its involved conditions have to be evaluated\(^2\). PoIPA language allows for specifying some multiple conditions, called predicates and defined as regular expressions involving the parameters of the actions and the attributes of users, resources, and environment. Thus, to get 100% coverage for a single PoIPA predicate, all combinations of true and false for parameters of the actions and for the attributes of the users, resources, and environment have to be considered. By iterating the process for every predicate, the 100% coverage of a PoIPA policy is reached when 100% coverage of its predicates is achieved.

**Testing framework**

Similarly to those generated for history-based testing, the test cases (requests) for continuous policy enforcement testing, consist of a sequence of commands for the PDP. In addition, they include a set of specific commands for changing the attributes values. This modification of attributes is performed during the execution of a test case and could invalidate the access rights causing the access revocation according to the PoIPA policy. More details about the strategy are in [17].

For checking the correctness of the PDP responses in the case of continuous policy enforcement testing an automatic oracle is implemented. The oracle analyzes the test case structure and extracts the sequence of commands included in the test case before a revocation command. Then evaluate the condition associated in the policy to the same revocation command. Finally the oracle decided the expected results by the evaluation of the predicate preceding the revocation command. In particular if the predicate is true:

- in case of a **before the access request evaluation**, the expected reply is a revocation command;

- in case of a **during the access execution** the expected reply is a revocation command;

\(^2\)Note that the term condition here does not refer to the environment conditions specified in the UCON model.
in case of an *after the access execution*, no revocation command is allowed; In case the evaluation of the predicate preceding the revocation command is false, the oracle simply ignores the PDP replies obtained after the revocation command. Whether an error or a delay of the replay occur, the PDP replay is considered erroneous, the test case execution is terminated and the overall verdict is fail.

The testing framework able to implement both history-based and continuous policy enforcement consists of the following components (see Figure 3.4):

**Fault Model Manager (FMM)** This component is involved in the testing of history-based access control. It manages a predefined collection of possible types of faults that can occur during the evaluation of a PoIPA policy due to incorrect implementation of commands, guards, or compositional operators.

**Policy Test Set Manager (PTSM)** This component collects the set of PoIPA policies useful for testing purposes. The policies can be given as an input by the user or taken from a predefined collection memorized in an internal dataset. In the latter case, a set of PoIPA policies are specifically conceived to highlight the functionality of the PDP and exercise the different available features. The PTSM is also in charge of the interaction with the PAP for the correct configuration of the PDP with the policy that is used for testing purposes.

**Faulty Policies Generator (FPG)** This component is involved in the testing of history-based access control. It takes as input a policy and the fault model and derives a set of faulty policies by seeding the faults defined in the fault model into the policy itself. Each of the faulty policies represents a faulty implementation of the PDP.

**Test Cases Generator (TCG)** This component derives the set of test cases for history based and continuous policy enforcement testing. In the former case, for each of the available policies (i.e. the policy and its faulty versions) TCG automatically derives the test cases in terms of (sequence of) access requests. Again, in case of continuous testing, TCG derives the sequence of access requests; in addition it interleaves them with specific commands called PIPCommand.

**Mutable Attribute Updater (MAU)** This component manages the attributes of the user, resource and environment. In particular, it simulates the PIP behavior by interacting with the PDP both for receiving
requests of attributes subscription and for updating the values of the attributes involved in a predicate of a revocation command during the test case execution.

**Test Driver (TD)** This component coordinates test execution. By collaborating with the TCG, it selects one by one the available test cases and, by simulating the PEP behavior, transforms each test case into an appropriate sequence of suitable commands. Moreover, during the execution of a test case, the TD interacts with the MAU by means of a PIPCommand so as to set the values of mutable attributes specified in the test case.

**Test Oracle (TO)** This component is responsible for the collection of the PDP responses caused by a test execution. TO also compares the obtained results with the correct authorization replies associated to each of the generated (set of) test cases. For this TO interacts with TCG for having the policy and the derived set of test cases. It is important to specify that in the current implementation only the functionality for the evaluation of the continuous policy enforcement test results is automated. We are evaluating and defining possible solutions for the analysis of history-based test results.

For a more detailed description of the components in the list above, we refer the interested reader to [17].

**Conclusions**

We proposed a framework aimed at the automated generation and execution of test cases, for testing of the PolPA-based PDP implementation. It is specifically conceived for addressing the dynamic runtime behavior of the PDP due to the mutability of attributes. The framework implements standard technique for multiple condition coverage and integrates a methodology for simulating all the possible ways in which attributes can change during the runtime execution of PolPA commands. For testing the continuous policy enforcement functionality, we also provided a fully automated facility for the derivation and evaluation of the expected test results. We have evaluated the testing framework using on a real policy and PDP implementation and we have showed the effectiveness of the proposed approach in revealing faults. We refer the interested reader to [17].

**3.2 Runtime verification**

**3.2.1 Quantitative controllers specified with generalized process algebras**

Security is often regarded as a binary concept, as it usually strictly depends on satisfaction of a boolean policy. However, in a broader sense, security has several dimensions, e.g., secrecy, anonymity, enforce-ability, availability, risk, trust and so on and so forth. These dimensions give rise to a multi-dimensional solution space for the construction of a “secure program”. On top of this, functional requirements add to the picture costs, execution times, rates, etc. All these dimensions make it meaningless to talk about a “secure” program, shifting the focus to the definition of a globally optimal solution, which easily fails to exist.

Here, we consider security as a quantitative, multi-dimensional measure of a system, and investigate possible answers to the question of what it means to enforce a security policy in this new setting. Our main actors will be controllers that constrain targets to obey to policies, using enforcement strategies. The ultimate goal of the proposed research direction is to define quantitative evaluation of enforcement strategies, that would provide analysis tools to compare and select different controllers, according to several metrics.

**Quantitative processes**

In a quantitative process, observable transitions are labelled with some quantity, denoting a cost or a benefit associated to a step in the behaviour of a system. We use semirings to model two fundamental modes of composing observable behaviour, either by combination of different traces, or by sequential composition. As a syntax to describe such behaviour, we adopt Generalized Process Algebra (GPA) from [22]. We first provide the definition of a semiring.
Definition 3.1 A semiring $\mathbb{K} = (K, +, *, 0, 1)$ consists of a set $K$ with two binary operations $+, *$, and two constants $0, 1$, such that $+$ is associative, with neutral element $0$; $*$ is associative, with neutral and absorbing elements $1, 0$; $*$ distributes over $+$.

Examples of semirings are natural numbers, the positive real numbers, boolean algebras, that may denote, e.g., discrete and continuous time, costs, lattices of values.

Semirings have a partial order $\sqsubseteq$, such that $k_1 \sqsubseteq k_2$ if, and only if $k_1 + k_2 = k_2$. Intuitively, $\sqsubseteq$ indicates preference, that is, $k_1 \sqsubseteq k_2$ can be read as $k_2$ is “better” than $k_1$. Sometimes, the $+$ operation is idempotent, and it extends to an operation $\sum_{\mathcal{S}}$ defined over an arbitrary, possibly infinite subset $\mathcal{S}$ of $K$.

Furthermore, semirings are composable in various ways. Indeed, for instance, the cartesian product of semirings is a semiring; thus, multi-dimensional notions of cost can be modelled. The partial order of values in the product does not prioritize dimensions. Further composite semirings exist, such as the lexicographic semiring, the expectation semiring, etc. When the specific composition operator is not of values in the product does not prioritize dimensions. Further composite semirings exist, such as the lexicographic semiring, the expectation semiring, etc. When the specific composition operator is not relevant, we shall indicate a composite semiring by $\mathbb{K}_1 \circ \ldots \circ \mathbb{K}_n$.

In quantitative process algebras, transitions are labelled by pairs $(a, x)$ where $x$ is a quantity associated to the effect $a$. We now define Generalized Process Algebra (GPA).

Definition 3.2 The set $\mathcal{L}$ of agents, or processes, in GPA over a countable set of transition labels $\text{Act}$ and a semiring $\mathbb{K}$ is defined by the grammar

$$A ::= 0 | (a,k)A | A + A | A||S A | X$$

where $a \in \text{Act}$, $k \in K$, $S \subseteq \text{Act}$, and $X$ belongs to a countable set of process variables, coming from a system of co-recursive equations of the form $X \triangleq A$. We write $\text{GPA}[\mathbb{K}]$ for the set of GPA processes labelled with weights in $\mathbb{K}$.

Process 0 describes inaction or termination; $(a,k)A$ performs $a$ with weight $k$ and evolves into $A$; $A + A'$ non-deterministically behaves as either $A$ or $A'$; $A||S A'$ describes the process in which $A$ and $A'$ proceed concurrently and independently on all actions which are not in $S$. All the action in $S$ are performed if and only if both the process perform the same actions in $S$ at the same time.

As we are dealing with run-time enforcement, we work with traces, or paths, of processes. The notion of path in is the usual one, that is, a sequence $(a_1, k_1) \ldots (a_n, k_n)$ leading from process $A$ to process $B$. We call $\mathcal{T}(A)$ the set of paths rooted in $A$. Given a path $(a_1, k_1) \ldots (a_n, k_n)$, we define its label $l(t) = a_1 \ldots a_n$, and its run weight $|t| = k_1 \ldots * k_n \in K$. The first notion we define is that of valuation:

Definition 3.3 Call valuation of process $A$ the value $\|A\| = \sum_{t \in \mathcal{T}(A)} |t|$.

Quantitative monitor and controller operators

A system, hereafter named target, does not always come with the quantities we are interested in evaluating, and might even be not labelled at all. Hence, in the most general case, the security designer must provide a labelling function $\lambda : \text{GPA}[\mathbb{K}_1] \rightarrow \text{GPA}[\mathbb{K}_2]$, such that given any process $A$ labelled in $\mathbb{K}_1$, $\lambda(A)$ represents the process $A$ labelled with a quantity in $\mathbb{K}_2$. A simple example is the function $\lambda_v$, which assigns any transition with the value $v \in \mathbb{K}_2$, thus erasing any previous quantity.

In practice, the responsibility of measuring a particular aspect is often delegated to a monitor, which probes the system and indicates the weight of each operation. In terms of security, a monitor is usually passive, i.e., it does not effectively modify the behaviour of the considered target. This means that it does not prevent violation of a security policy. On the other hand, a controller is able to modify the behaviour of a target in order to guarantee security requirements. A security monitor and a security controller are often merged into a single entity, responsible both for deciding whether an action would violate the policy and what corrective action should be taken if necessary. We propose here to make an explicit distinction between these two processes and to extend the monitoring to measures other than security. We first discuss monitors and then controllers.

Intuitively, a monitor measures a quantity not already present in the monitored target. Since the target might be already equipped with some quantities, coming for instance from another monitor, we need to merge the quantities from the monitor with those of the target.
Table 3.2: Semantic rules for quantitative control operators.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(E \circ a, k) \rightarrow (E \circ a, k) \rightarrow F \circ a, k \rightarrow F' \circ a, k \rightarrow F'')$ (A)</td>
<td>A merged process can only move on when both of its components can move on with the same action. We are now able to define a monitor, which is a process that can be composed with a target without affecting its behaviour.</td>
</tr>
<tr>
<td>$(E \circ a, k) \rightarrow (E \circ a, k) \rightarrow F \circ a, k \rightarrow F' \circ a, k \rightarrow F'')$ (S)</td>
<td>Given any process $A$ labelled with $K$, a process $A'$ labelled with $L$ and a composition operator $\circ$, we write $A \circ A'$ for the merged process, defined as:</td>
</tr>
<tr>
<td>$A \circ A'$</td>
<td>$A \circ A'$</td>
</tr>
</tbody>
</table>

A merged process can only move on when both of its components can move on with the same action. We are now able to define a monitor, which is a process that can be composed with a target without affecting its behaviour.

**Definition 3.4** Given a composition operator $\circ$ and a process $A$, a process $M$ is a monitor for $A$ if and only if $\{t(|t| t \in T(A \circ M)) = \{t(|t| t \in T(A))$.

Given any process $A$ labelled with $K_1$, any monitor $M$ for $A$ labelled with $K_2$ and any composition operator $\circ$, we can define the labelling function $\lambda : GPA[K_1] \rightarrow GPA[K_1 \circ K_2]$ as $\lambda(A) = A \circ M$. Clearly, finer-grained approaches can be used to monitor a security policy. Note that a monitor is only one possible way to build a labelling function $\lambda$. Although monitors are expressive enough for the examples we consider in this paper, more complex labelling functions may also be of interest.

The role of a monitor is to detect a policy violation, and not to prevent a target system from doing so. For this reason it can be used, for instance, for directly evaluating a security policy $P$ as a value on each transition of a target process.

A controller $E$, just like a monitor $M$, follows target actions step by step. The difference is that $M$ observes target actions, labelling them with $true$ when they obey to the policy $P$ or $false$ when they attempt to violate $P$. On the contrary, the controller can decide not only to accept but also to change target traces. The resulting process is the controlled process $E \circ F$, where $F$ denotes the target system, following the semantics given in Table 3.2.

Intuitively speaking, each rule corresponds to a different controlling behaviour. The alphabets of $E$, $F$, and of the resulting process $E \circ F$ are different, as $E$ may perform control actions that regulate the actions of the target $F$, and moreover the resulting process $E \circ F$ may perform internal actions, denoted by $\tau$, as a consequence of suppression. From now on, we will let $Act$ be the alphabet of (the GPA describing) $F$. The alphabet of $E$ consists of symbols of the form $a, \Xi a, \Xi a, b, for a, b \in Act$, denoting respectively the actions of acceptance, suppression, and insertion; the alphabet of $E \circ F$ is $Act \cup \{\tau\}$.

The acceptance rule (A) constrains the controller and the target to perform the same action, in order for it to be observed in the resulting behaviour; the observed weight is the product of those of the controller and the target. Given two processes $A$ and $B$, the semantics of truncation is equivalent to that of CSP-style parallel composition of $A$ and $B$, where synchronisation is forced over all actions of the two processes.

The suppression rule (S) allows the controller to hide actions of the target. The target wants to perform the action, but the action is not performed by the controlled entity and the observed result is a $\tau$ action, with the weight calculated as the product of the suppressing and the target action.

Finally, the insertion rule (I) describes the capability of correcting some bad behaviour of the target, by inserting another action in its execution trace. The weight of insertion is only the weight provided by the controller; this accounts for the fact that the target does not perform any action, but rather stays in its current state, as in [9].

**Evaluation and ordering of controller strategies**

In order to evaluate a given controller against a given target, different labelling functions can be used, and the actual order can have an impact on the global valuation. Indeed, in order to be consistent with the
semantic rules, if the controlled target is labelled with some weight, the controller needs to be labelled with compatible weights. For instance, given a target monitored by \( M_F \), the controller must be labelled with \( K_F = (\{ \text{true, false} \}, \lor, \land, \text{false, true}) \).

Clearly, these approaches are not exclusive, and it might be valuable to monitor both how many times the target tried to violate the policy and whether the controlled target violates the policy. In some cases, it might be desirable to monitor both the controller and the target independently. For instance, the controlling actions can be associated with a notion of cost [35]. Let us now introduce the notion of matching operator \( \rtimes \) :

\[
E \rtimes F = \lambda_T(\lambda_E(E) \rhd \lambda_F(F))
\]

where \( \lambda_E \) labels the controller, \( \lambda_F \) labels the target and \( \lambda_T \) labels the controlled target. It is possible to provide a classification for any considered semiring.

**Definition 3.5** Given a target \( F \) and a matching operator \( \rtimes \), a controller \( E_2 \) is better than a controller \( E_1 \) with respect to \( F \), and in this case, we write \( E_1 \subseteq_{\rtimes,F} E_2 \), if and only if \( \|E_1 \rtimes F\| \subseteq \|E_2 \rtimes F\| \).

This definition does not directly depend on the semiring used to quantify the controlled target, and it is therefore possible to use the same definition to say that a controller is better than another one with respect to a security monitor, a cost monitor or any other measure. Since it might be that a controller is better than another one for a specific target, and that the converse holds for some other target, we introduce a stricter ordering, where the comparison is performed over all possible targets.

**Definition 3.6** Given two controllers \( E_1 \) and \( E_2 \) and a matching operator \( \rtimes \), we say that \( E_2 \) is always better than \( E_1 \), and in this case we write \( E_1 \subseteq_{\rtimes} E_2 \), if and only if \( E_1 \subseteq_{\rtimes,F} E_2 \), for any target \( F \). In addition, if \( E_1 \subseteq_{\rtimes} E_2 \) and there exists at least one target \( F \) such that \( \|E_1 \rtimes F\| \subseteq \|E_2 \rtimes F\| \), we say that \( E_2 \) is strictly better than \( E_1 \), and write \( E_1 \subset_{\rtimes} E_2 \).

Since each individual trace can be represented as a target, it implies that the valuation of \( E_1 \) should be lower for every possible trace. Hence, this definition identifies the cases where a controller strategy is always better than another one.

In some cases, controllers can be incomparable. The choice of the controlling operators can have an impact on the overall evaluation. Moreover, other dimensions can easily be included within our framework, with the intuition that the more accurate the quantification of the controlled system, the more informed is the security designer to choose a particular controller. In general, there might not be a strictly best strategy. In some cases, it might be possible to define an optimal strategy, which is best on average.

Another possibility is to prioritize one dimension over another (depending on the order of the components in the lexicographic order itself). According to which dimension is prioritized, we are able to classify controllers into categories. Examples of such categories are the following ones. We have a secure controller when the controllers are ordered based on their security, that is, how many traces are correct. This measure depends on the number of traces \( t \in T(E) \) such that \( |t| = \text{true} \). Optimal controllers are those that never violate the policy in conjunction with any target. On the other hand, a controller can be said to be economical when the priority is given to the dimension of cost. Hence, the obtained order on controllers considers the cost on each trace, and the optimal controller is the one that costs less.

### 3.2.2 Synthesizing a mediator satisfying policy constraints

The composition of web services that meet a desired functionality whilst satisfying a given security policies represents a major challenge. Expressing security policies with automata has been investigated in many works, e.g., [33, 53, 74]. The policies that might be imposed on web services' behaviour deal with access control, inputs or actions sequencing and allowed communications between agents. More precisely, the following problem is addressed here: given a client specification \( C \), an available community of services \( S_1, \ldots, S_n \), and a policy \( P \), synthesize a mediator \( M \) that suitably directs the actions of the client to the services whilst respecting the policy \( P \).

A new tentative approach is introduced where all the agents \( C, S_1, \ldots, S_n \) and the policy \( P \) are expressed as communicating guarded variable automata (CGVA) [10]. The idea is that when designing a mediator for scheduling the actions of the services one has to consider security policies too. A policy
Let us introduce the formal definition of CGVAs. We denote by $T$ the set of symbols in a set of function symbols $\Sigma$ and a set of variables in $X$. We shall denote by $T(\Sigma, X)$ the set of formulas built from the symbols in $\Sigma$ and the variables in $X$. We shall denote by $\overline{T}(\Sigma, X)$ the set of formulas built from the symbols in $\Sigma$ and the variables in $X$. If $t \in T(\Sigma, X)$ then $!t$ (resp. $?t$) denotes sending (resp. receiving) of message $t$. A substitution is a mapping from $X$ to $T(\Sigma, X)$. The set $\mathcal{G}$ of guards over $T(\Sigma, X)$ is defined inductively to be either $true$ or an equality or an inequality between terms or a conjunction of guards.

**Definition 3.7** A CGVA is a tuple $\mathcal{A} = (\Sigma, X, Q, Q_0, \delta, F, \kappa)$ where $\Sigma$ is a finite set of function symbols, $X$ is a finite set of variables, $Q$ is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $\delta : Q \times (\Sigma_A \times T(\Sigma, X) \times \mathbb{G}) \rightarrow 2^Q$ is a transition function where $\Sigma_A$ is a finite set of constants, $F \subseteq Q$ is a set of accepting states, and $\kappa : X \rightarrow 2^Q$ is a refreshing function.

Let $a$ be a constant in $\Sigma_A$, $t$ a term in $T(\Sigma, X)$, and $g$ a guard in $\mathcal{G}$. The intuition behind a transition $q \xrightarrow{(a,t),g} q'$, is the following: instantiate all the free variables in $t$ so that the guard $g$ holds, and change the state to $q'$, then refreshes all the variables in $\kappa^{-1}(q')$. The set of finite sequence of terms recognized by a CGVA $\mathcal{A}$ is denoted by $L(\mathcal{A})$.

Several types of security policies can be expressed with CGVAs. The CGVA in Figure 3.5 enforces a policy stating that “a private message of the Client $c_1$ is never sent to the Client $c_2$, and vice versa”. In this example, the action $(S, !m)$ means that the service $S$ sends the message $m$. We can also specify that the variables $m$ and $t$ are refreshed in the state $u_0$.

**Mediator synthesis with policy enforcement**

The standard simulation relation on finite automata can be generalized to CGVA and used to formalize that client requests are always satisfiable by an available service (both being specified by CGVAs). We call this relation a communicating simulation for CGVAs, or $c$-simulation for short, and explain it briefly. The $c$-simulation is a binary relation over configurations. A configuration is a pair $(q, \gamma)$ where $\gamma$ is a substitution and $q$ is a state. Assume a client configuration $(p, \gamma_c)$ and a transition labeled by $!t$ outgoing from $p$, in which the client instantiates all the free variable of $\gamma_c(t)$ by a substitution $\sigma_t$. Assume a service configuration $(q, \gamma_s)$. Hence the Client transition above should be matched by a service transition labeled by $?u$ so that the service has to compute a substitution $\sigma_u$ such that $\sigma_t(\gamma_c(t)) = \sigma_u(\gamma_s(u))$. On the other hand, a client transition labeled by $?t$ can be simulated by a service transition labeled by $!u$, provided that
the service computes two substitutions $\sigma_t$ and $\sigma_u$ such that $\sigma_t(\gamma_c(t)) = \sigma_u(\gamma_s(u))$. In a formal definition of \cE\-simulation we should also take into account the refreshing of variables.

Let $C$ be a client specification, $S_1, \ldots, S_n$ be $n$ services, each of which is a CGVA. We denote by $\otimes$ the asynchronous product of CGVAs. A mediator allowing the communication between $C$ and $S_1, \ldots, S_n$ is a GVA $M(C,S_1, \ldots, S_n)$ that recognizes the successful communication sequences $s = t_1\tau t_1 \ldots t_n\tau t_n$ between $C$ and $S_1 \otimes \ldots \otimes S_n$. More generally, the problem of mediator synthesis under policy enforcement can be stated as follows:

**Input:**  Client $C$, Services $S_1, \ldots, S_n$, and Policy $P$, all specified by CGVAs.

**Output:**  A mediator $M(C,S_1, \ldots, S_n)$ satisfying $L(M) \cap L(P) \neq \emptyset$.

Solving the composition synthesis problem above, relies first on the construction of a two-player game $G(C,S_1 \otimes \ldots \otimes S_n)$ played between Abelard (or Falsifier) and Eloise (or Verifier), such that $C \preceq (S_1 \otimes \ldots \otimes S_n)$ if and only if Eloise has a winning strategy in that game. Then a winning strategy for Eloise in the game $G(C,S_1 \otimes \ldots \otimes S_n) \times P$ yields a mediator that respects the policy $P$. Although the composition synthesis is undecidable in general for CGVAs, it is decidable for subclasses of practical interest, for instance when messages are flat terms.

**Conclusions**

A new class of automata has been introduced in order to express services and policies and a mediator synthesis algorithm has been designed. This approach should be extended in future works to more expressive policies and timing constraints.
4 Quantitative Methods

In line with current research trends, we broaden our focus from pure security metrics to a more general view on quantitative aspects of security. Here, the first line of research is an investigation of different vulnerability predictors, applied in a case study to the Firefox web browser. The second line of research focuses on modeling an attacker who dynamically adapts her behaviour as her knowledge about a system’s vulnerabilities increases.

4.1 Vulnerability prediction in Firefox

In Deliverable D9.3 [83], we have presented a vulnerability prediction technique based on text mining. According to that technique, each source file of an application is parsed, e.g., with a lexical analyzer, and tokenized into its elementary words: the language keywords, the variable and function names, the mathematical operators, and so on. In the case of C/C++ source code, the comments, the pre-processor directives, and the literals are ignored.

At the end of the parsing process, each source file is represented as a stream of words, which is then converted to a vector of features using the Weka tool. The features consist of the unique words appearing in a file and their corresponding frequencies, i.e., the number of times a certain word is used in a source file. In the literature, this representation is referred to as a ‘bag of words’. The number of unique words in an application (i.e., the vocabulary) can be quite high. For instance, an application like Mozilla Firefox contains over 280 thousands words. Therefore, only the subset of the most frequent words is retained. In Weka, the default is to retain up to 1000 words (per class).

Our prediction technique uses machine learning algorithms like the Random Forest ensemble learner in order to predict which files are vulnerable, given its set of features (words and frequencies). To this aim, each feature vector needs to be labeled as belonging to a vulnerable or clean file (binary classification). In previous work, we used the vulnerability warnings generated by a static analysis tool as the ‘oracle’ to label the application files. That choice was mandated by the lack of vulnerability data for the class of applications we were analyzing (Android apps written in Java). In this work, instead, we focus on the Firefox browser, whose vulnerabilities are tracked in the NVD (National Vulnerability Database) and MFSA (Mozilla Foundation Security Advisories) databases. Therefore, we use the information in those databases to assess whether a source file contains security vulnerabilities. If one or more issues are reported for a certain file, the file is labeled as vulnerable.

We have considered 9 consecutive releases of Firefox, i.e., from version 4 to version 12. We have built a prediction model with the training data (features and classifications) coming from version 4. We have used the model to predict the vulnerability status of the subsequents versions. The predictions are compared to the actual status of each file (vulnerable or clean) and several performance indicators are computed. As a side note, we merge the files that share the same file name and have different extensions, namely, the header (e.g., .hpp) and the body (e.g., .cpp) files of a C++ class. This approach is in line with the work of Neuhaus et al. [61]. We dub this merged unit of code as ‘component’.

To summarize, the contribution of this work with respect to our previous work is the validation of our technique on a different class of applications (web browser), a different programming language (C/C++), and using vulnerability data coming from authoritative sources (NVD/MFSA). This work confirms that our technique works very well and its performance outclasses the state of the art. The technique could be used to identify the components of an application that deserve special attention from the quality assurance team, e.g., to prioritize the testing plans and the code review activities.

Words as predictors

The intuition is that the features extracted from the bag-of-word representation retain some software properties of the code like the size (e.g., via the number of features) or the cyclomatic complexity (e.g., via the presence of many features related to control flow keywords). These property have a proven influence on the security of a software code base. It has been shown that complexity metrics have some predictive power with respect to vulnerabilities, as more complex files contain more vulnerabilities [77]. However, our technique does not postulate which characteristics of the code are more influential. Nevertheless, the
Table 4.1: Predicting future Firefox versions. The model produces excellent performance results that outperform the state of the art.

<table>
<thead>
<tr>
<th>Version</th>
<th>R</th>
<th>FI</th>
<th>P</th>
<th>A</th>
<th>FPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>99</td>
<td>18</td>
<td>16</td>
<td>85</td>
<td>16</td>
</tr>
<tr>
<td>6.0</td>
<td>98</td>
<td>19</td>
<td>16</td>
<td>84</td>
<td>16</td>
</tr>
<tr>
<td>7.0</td>
<td>97</td>
<td>19</td>
<td>15</td>
<td>84</td>
<td>17</td>
</tr>
<tr>
<td>8.0</td>
<td>98</td>
<td>19</td>
<td>14</td>
<td>84</td>
<td>17</td>
</tr>
<tr>
<td>9.0</td>
<td>98</td>
<td>19</td>
<td>13</td>
<td>83</td>
<td>17</td>
</tr>
<tr>
<td>10.0</td>
<td>98</td>
<td>17</td>
<td>12</td>
<td>85</td>
<td>16</td>
</tr>
<tr>
<td>11.0</td>
<td>96</td>
<td>17</td>
<td>13</td>
<td>85</td>
<td>15</td>
</tr>
<tr>
<td>12.0</td>
<td>93</td>
<td>17</td>
<td>13</td>
<td>85</td>
<td>15</td>
</tr>
<tr>
<td><strong>Average (%)</strong></td>
<td><strong>97</strong></td>
<td><strong>18</strong></td>
<td><strong>14</strong></td>
<td><strong>84</strong></td>
<td><strong>16</strong></td>
</tr>
</tbody>
</table>

The prediction model can be analyzed in order to extract the list of features that shape the decision process of the prediction model. These features provide insights on the code patterns that are likely to introduce vulnerabilities.

**Labeling of Firefox components**

In order to label the components of the Firefox application as either vulnerable or clean, we follow two steps:

- First, we collect the vulnerabilities for individual versions of Firefox.
- Second, we map the vulnerabilities to the components in the code base that are potentially responsible for the vulnerabilities.

**Acquiring vulnerabilities for Firefox** We applied a repository mining technique to identify Firefox’s security bugs. The repository mining technique has been introduced by Sliwerski et al. [79] and adopted by [61,76]. The commit logs of the code base repository are parsed and the bug identifiers are extracted. If an extracted bug ID is also mentioned in a MFSA or NVD entry, the bug is marked as a vulnerability. The Mozilla Foundation Security Advisory (MFSA, [http://www.mozilla.org/security/announce/](http://www.mozilla.org/security/announce/)) is the vendor advisory for announcing security issues and their fixes for all Mozilla’s products. The National Vulnerability Database (NVD, [http://nvd.nist.gov/](http://nvd.nist.gov/)) is a third-party database recording and publishing known vulnerabilities for a vast range of applications.

We also need to relate the bug ID to the affected product versions. To this aim, we look into the following sections of the vulnerability advisories:

- An MFSA entry refers to the versions of Firefox in its fixed releases section.
- An NVD entry mentions the version of Firefox in its vulnerable versions section.

**Mapping vulnerabilities to components** For each commit mentioning a security-related bug ID, we collect the files that are affected by changes. These files are marked as responsible for the vulnerability in all the versions that are affected by the vulnerability (as per the NVD/MFSA information) and contain those files. If a component (in a version) is responsible for one or more security-related bugs, it is labeled as vulnerable (in that version).

**Performance indicators**

As mentioned earlier, each component is predicted by the model as either vulnerable (1) or clean (0). The prediction is compared to the observed value, i.e., the real value that is based on the NVD/MSFA databases. Accordingly, the prediction can be a true positive (TP, the component is both predicted and observed as vulnerable), a true negative (TN, the component is clean for both the predictor and the
vulnerability databases), a false positive (FP, the component is predicted as vulnerable albeit it is not), or a false negative (FN, the component is predicted as clean but in reality it is not). For each release we compute the total number of components that are predicted as TP, TN, FP, and FN. These values are then used to compute the performance indicators of the prediction model.

In the literature on Firefox vulnerability prediction [76], the most important indicators are the recall ($R$) and the file inspection rate ($FI$). Recall (sometimes called probability of detection, true positive rate, or sensitivity) represents the probability that a vulnerable file is successfully classified as such. Its definition is as follows:

$$R = \frac{TP}{TP + FN} \quad (4.1)$$

A high value of recall is desirable because it means that the results returned by the prediction model are very complete and the risk of not scrutinizing a vulnerable component is minimal.

The file inspection rate is the percentage of components that the quality assurance team is supposed to scrutinize if the predictions of the model are followed up. Its definition is as follows:

$$FI = \frac{TP + FP}{TP + TN + FP + FN} \quad (4.2)$$

For instance, $R = 80\%$ and $FI = 20\%$ means that 80 percent of the vulnerable components are found by inspecting 20 percent of the total number of components in the application.

Other performance indicators often used in the related work are precision ($P$), accuracy ($A$), and the false positive rate ($FPR$). We report these indicators for completeness and in order to facilitate the comparison of our results with the results of other studies.

Results

Table 4.1 and Figure 4.1 summarize the results of our experiment. In particular, the numbers refer to the average performance (out of 10 runs) of the Random Forest algorithm. Similarly to [76], we have applied an under-sampling technique on the training set.

On average, across all the tested version, the recall ($R$) is 97% and the file inspection ($FI$) is 18%. These performance results are remarkable. As a reference, the best performance found in the related work corresponds to a recall of 88% and a file inspection of 17%. Hence, our technique has a recall that is about 10 percent points better, while preserving the same (low) file inspection rate. In practice, many more vulnerable components are found with the same inspection effort. Furthermore, the metrics used in the related work are much more costly to collect than ours, as they require the tracking of the developers’
activity, as well as the monitoring of the changed lines of code across several versions. Note also that the performance is rather stable over time: the first ‘bigger’ drop in performance takes place in version 12, i.e., 13 months after the model has been built in version 4.

Conclusions

In summary, we have shown the feasibility of a vulnerability prediction technique based on text mining. The technique has been validated in two different contexts: mobile applications written in Java (last year) and web browser written in C++ (this year). The results show that the technique yields dependable results.

4.2 Modelling adaptive attacker’s behaviour

An optimal distribution of security investments requires an understanding of how attackers can penetrate the system. In other words, we should understand possible behavior of an attacker. This behaviour determines how attackers choose the attacks to follow. Thus, we would like to consider the information about the system available to the attacker and how the attacker collects this knowledge.

Most methods for the analysis of security of computer systems (e.g., networks, Cloud, etc.) consider attackers as omniscient entities which know all weaknesses of a computer system [47, 75]. In addition, attackers are frequently assumed to make only right decisions during an attack and to exploit only the best possible way for the attack. In contrast, descriptions of real complex attacks (e.g., [59]) show that attackers have limited knowledge of a target system and explore the system step by step during the attack. Attackers make mistakes in their reasoning about the system, and search for alternative ways to compromise the system when the initially selected attack fails. This means that the model of powerful attacker does not provide a realistic description of a situation, but prepares for a worst case scenario. In reality, security teams have a limited budget and would like to concentrate on the most important security issues that can be solved within a budget.

In this work, we strive for a more refined attacker model introducing the attacker’s view of a system, which is sometimes different from the real system. This view drives the actions of the attacker depending on her knowledge and resources. Moreover, in our model an attacker may give up on her current attack and follow an alternative attack path. Thus, we get information about alternative attacks (and their probability of occurrence) the attackers will launch against the system. We use Markov Decision Processes (MDP) to model the attacker’s behaviour as the method for the selection of attack steps.

Modelling the attacker behaviour

We consider a computer system as an attack graph G that represents the ways to compromise the system [47, 75]. A node \( s_i \in S \) of the attack graph denotes a successfully exploited vulnerability and an edge \( a_{ij} \in A \) denotes further possible exploitation of vulnerability \( s_j \) after the previous exploitation of vulnerability \( s_i \). Thus, successful exploitation of vulnerabilities leads an attacker to new states with new privileges. There are several methods for automated construction of attack graphs [75, 88].

We separate the real system and the attacker’s belief about the system. The attacker’s knowledge about the system determines the set of vulnerabilities that he believes to be present in the system. This set of vulnerabilities is further reduced depending on attacker’s skills and tangible resources. Finally, the attacker has her own view (a graph \( G_X = (S_X; A_X) \)) of the system.

We assume that the system behaves probabilistically. We associate the probability \( Pr_{ij} \) with the system transition from state \( i \) to state \( j \) in response to an attacker’s action. For the attacker this probability is:

\[
Pr_{ij} = Pr^p_{ij} \times Pr^{exp}_{ij}
\]  

(4.3)

where \( Pr^p_{ij} \) is the probability that the vulnerability \( j \) is present in the system and \( Pr^{exp}_{ij} \) is the conditional probability that the vulnerability may be successfully exploited in case it exists in the system.

Finally, for all states we assign a terminal reward \( r_j \in R \) the attacker gets when she reaches the state \( s_j \). In our model, we assign terminal rewards to the goal states only (all other states have terminal reward equal to 0). We also may consider the instant reward \( r_{ij} \in R \), the attacker gets when she successfully
Algorithm 2 Computation of a deterministic policy

\[ t = N \]

for \( j, s_j \in S \) do
\[ u_j^N = r_j \]
end for

while \( t > 1 \) do
\[ t = t - 1 \]
for \( k, s_k \in S \) do
\[ u_k^t = \max_{a_{ki} \in A_k} \left\{ r(a_{ki}) + \sum_{a_{ij} \in A_i} \Pr_{ij} \cdot u_j^{t+1} \right\} \]
\[ \pi_k = \arg \max_{a_{ki} \in A_k} \left\{ r(a_{ki}) + \sum_{a_{ij} \in A_i} \Pr_{ij} \cdot u_j^{t+1} \right\} \]
end for
end while

executes an action \( a_{ij} \). In many cases all instant rewards are equal to 0. Note, that a terminal reward is consumed only when an attacker is in the corresponding state, when instant rewards are aggregated while an attacker moves from a state to a state.

The algorithm for the computation of optimal deterministic policies is the backward induction [70] (see Algorithm 2). Backward induction is an algorithm for the computation of a policy of a finite-horizon discrete-time Markov decision problem. Finite horizon means that the number of decision epochs for the attacks is bounded which is true because the attacker never has infinite time for performing the attack. We consider \( N \) epochs in Algorithm 2. Let \( \Pi \) be a set of policies \( \pi \in \Pi \). A policy \( \pi_k \) refers to the action \( a_{ki} \), which leads to that maximisation of the expected total reward for the attacker. Now, the attacker should take the initial state \( (s_0) \) and follow the suggestion \( (\pi_0) \), leading it in a new state, where the attacker should again select the suggested action. And so on, until she reaches its goal. The attacker does not obtain the maximal reward each time. However, in case of several attacks (e.g., within an attack profile) the average reward will be maximal.

For every epoch \( t \) we determine the utility value \( (u_k^t) \) and the most profitable action to take \( \pi_k \) for every state. We use \( A_i \) as a set of edges outgoing from a state \( s_i \). Note, that for look for the most profitable action to take in state \( s_k \), we take the action, for which the further reward is maximal (e.g., \( \arg \max \)), not the value of the reward itself (not \( \max \)).

We modify the behaviour of the deterministic attacker so that she may reconsider her course of action when she cannot complete her current attack path. We assume that the attacker sets \( \Pr_{ij}^P = 0 \) (and thus \( \Pr_{ij} = 0 \)) when she cannot complete an attack step \( a_{ij} \) and understands that the vulnerability is absent from the system. In addition, the attacker sets \( \Pr_{xj}^P = 0 \) for all other edges entering \( s_j \) from all states \( s_x \). Then, the attacker uses the Algorithm 2 to compute a new attack policy using the updated attack graph and the amount of decision epochs left after the initial part of the attack.

Adapted attacker behaviour

The attacker sets \( \Pr_{ij}^P = 1 \) and \( \Pr_{ij} = \Pr_{ij}^{exp} \) if she understands that the vulnerability exists in the system as a result of the unsuccessful attempt of the attack step. This may happen because the attacker simply makes an error in executing an action, but gets an evidence that the vulnerability exists. Then the attack policy is recomputed according to Algorithm 2 with the rest of the decision epochs. If the attacker successfully exploits the vulnerability \( s_i \) she adds edges \( a_{0j} \) and sets \( \Pr_{0j} = \Pr_{ij} \) for all states \( s_j \) reachable from \( s_i \) in one step. This modification is required to remember the privileges gained by the attacker for future adjustments in her attack policy.

We see that for our analysis we need almost the same information required for other methods considering the worst case scenario [47,75]. The algorithm needs the attack graph and sets of probabilities and rewards. As a result the algorithm provides the probabilities for selecting one or another branch in the graph, i.e., the probability to execute one or another attack. We also may consider different strategy for the selection of an alternative path, e.g., an attacker tries several attempts before deciding to give up and
try to compromise another vulnerability (see [46] for details), but in this case the amount of attempts has
to be provided.

Example

We consider a computer system in a hospital. The hospital stores the data with the patients’ health
information in an online database service to be accessed by doctors. A competitor medical company
would like to compromise the hospital by stealing and releasing the sensitive information. The competitor
hires professional hackers to attack the server where the database is installed. The server operates
FreeBSD 7 and MySQL 5. The database is managed by an administrator who uses a local workstation
operated by Linux Mint 12 with Pidgin Messenger installed. Moreover, the administrator manages the
database from her home laptop using a VPN connection to the workstation. The laptop runs Windows 7,
Chrome browser, and TUKEVA Password Reminder. The whole system and a possible attack graph are
depicted in Figure 4.2.

In our example, the attacker has N decision epochs and gets terminal rewards (10K) only if she reaches
states 3, 7, 8, i.e., \( r(s_3) = r(s_7) = r(s_8) = 10 \) and all other terminal rewards equal to 0. Instant rewards
also equal to 0. For the attack graph presented in Figure 2, the policy is \( \pi_1 = a_{08} \) at the initial state during
the first decision epoch. Suppose, the action is unsuccessful because the vulnerability has timely patched
by the administrator. The attacker sets the probability \( Pr_{08} = 0 \), reconsiders her initial policies using N-1
decision epochs, and obtains new policy \( \pi_1 = a_{05} \).

Conclusions

This work presented our initial ideas on modelling the behaviour of an attacker. We think, such an ap-
proach is important in order to get a versatile analysis of our system and to protect it in the most efficient
way. In particular, we considered an attacker who does not know every detail about the system, but gains her knowledge step by step. In addition, we made the attacker more flexible, i.e., the attacker is able to re-consider her plans when the initial ones fail. For further details about this work, we refer the interested reader to [46].

In future work, we would like to improve our model by investigating further towards a probabilistic attacker, the attacker which does not behaves deterministically, and use this knowledge to find the probability of attack selection, which is required for a quantitative security assessment.
5 Relations to Other Work Packages

WP 9 is a transversal work package, which spans all phases of the SDLC. In this section, we describe concrete relations with other work packages and we provide pointers to the respective parts of the 3rd year deliverables of those work packages.

WP 2 – Integration of methodologies and tools in the WorkBench In Deliverable D9.3 [83], we presented a toolchain for transforming access control policies written in UWE (UML-modeling-language) to XACML and from XACML to the Formal Access Control Policy Language (FACPL). In Section 2.1.3 of this deliverable, we describe a new toolchain for testing XACML policies, where the UWE-to-XACML transformation is used as a front-end for the X-CREATE tool, which is also part of the NESSoS Service Development Environment (SDE).

WP 7 – Secure service architectures and design There are close connections to several contributions of Deliverable D7.4 [85]. The section on “Compositional verification of software architectures” is closely related to WP 9. In that section, a compositional method for the refinement of architectural designs is presented, which enables the decomposition of large designs into smaller components that can be verified and refined individually.

The section entitled “ActionGUI Toolkit and Applications” reports on the latest developments in the ActionGUI tool and on several applications demonstrating its usefulness. In Section 2.1.1 of this deliverable, we describe a component, OCL2FOL+, that plays a crucial role in enabling accurate formal analysis of ActionGUI models using standard reasoning tools for first-order logic.

Moreover, the section “An UML extension for modeling protection-specific security features for web applications” is related to the transformation of graphical UWE designs to TextualUWE, a Scala-based DSL, which is introduced in Section 2.1.2 of this deliverable.

WP 8 – Programming environments for secure and composable services The contribution “Synthesizing a mediator satisfying policy constraints” of the present deliverable is closely related to Task 8.2 of WP 8, entitled “Secure service composition for the Future Internet”.

The section entitled “Secure Navigation Paths” (SNP) of Deliverable D8.4 [86] has several connections with Task 9.2, “Assurance in Implementation” and with Section 2.1.2 of this deliverable. That work proposes a methodology for defining simultaneously RBAC and SNP policies for an application in UWE and for safeguarding them at runtime by a monitor, which also enforces navigational access control. Moreover, it provides means for automatically testing SNP policies for applications not running an enforcing monitor.

WP 10 – Risk and cost aware SDLC The section entitled “Test-Driven Security Risk Assessment” of Deliverable D10.4 [84] discusses several method for combining security testing and security risk assessment. In particular, an extension of CORAS for test-driven risk assessment is presented, where testing is used after the completion of the risk evaluation and the testing results are fed back to the risk analysis process for risk validation, before finalizing the risk analysis. Here, there is a clear connection with testing in WP 9.
6 Conclusions

In this deliverable, we have reported on further development of our methods and tools for security assurance for services that we have obtained during the third year of the NESSoS project. The methods and tools presented here cover all the major activities related to providing assurance in the design and implementation phases of the SDLC. They represent a well-balanced mix of continued and new strands of research. In particular, we have

- The generation of models from UML-based model-driven security design methods to enable formal analysis (Section 2.1, Task 9.1.2). This is essentially a continuation of the work reported in Deliverable D9.3 [83].

- Algorithmic verification of security protocols (Section 2.2, Task 9.1.3). The work on sound security protocol transformations is new but closely related to work on protocol verification reported earlier. We are considering a collaborative integration of these works.

- Testing of security protocols and access/usage control policies (Section 3.1, Task 9.2.2 and 9.3.1). The work on mutation-based security protocol testing and on testing of PolPA-based usage control systems continues earlier research (see Deliverable D9.3 [83]). Test case prioritization for XACML policies complements our earlier work on testing of XACML policies (see Deliverable D9.2 [81]). The work on the model-based vulnerability testing tool VERA is new.

- Runtime verification (Section 3.2, Task 9.2.3). The research on quantitative controllers specified in generalized process algebras is new. It combined runtime verification techniques with quantitative aspects (Task 9.4). The work on mediator synthesis is new but related to earlier work on the “Extension of CL-Atse with negative constraints handling” reported in Deliverable D9.3 [83].

- Quantitative aspects of security (Section 4, Task 9.4). The work on vulnerability prediction for Firefox continues earlier work on vulnerability prediction for Android applications (see Deliverable D9.3 [83]). The research on modelling the adaptive attacker’s behaviour is new.

The third year was also marked by the emergence of a number of new collaborations between project and associated partners as well as the deepening of existing collaborations.

- A group of five researchers from IMDEA has visited ETH during six months in order to deepen their collaboration on both theoretical aspects of model-driven security as well as on the ActionGUI tool development (see Section 2.1.1 and Deliverable D7.4).

- LMU and CNR have integrated their tools for the transformation of access control policies specified in UWE and the X-CREATE XACML policy testing tool (Section 2.1.3).

- ETH and SIEMENS have collaborated on the model-based vulnerability testing tool VERA (Section 3.1.2).

- CNR and associate partner University of Luxembourg have worked together on the prioritization of test cases for XACML policy testing (Section 3.1.3).

- KUL and UNITN have combined their expertise in their work on vulnerability prediction in Firefox (Section 4.1).

For the remainder of the project we plan to further extend and integrate our methods and tools and to continue advancing the state-of-the-art in all the major assurance activities of WP9.
A NESSoS WP 9 Third-year Publications


Bibliography


