Network of Excellence

Deliverable D9.3

Initial Prototypal Support for Security Assurance for Services
<table>
<thead>
<tr>
<th>Deliverable Number</th>
<th>D9.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title of Deliverable</td>
<td>Initial Prototypal Support for Security Assurance for Services</td>
</tr>
<tr>
<td>Nature of Deliverable</td>
<td>R</td>
</tr>
<tr>
<td>Dissemination Level</td>
<td>Public</td>
</tr>
<tr>
<td>Internal Version Number</td>
<td>3.0.1 (SVN Revision No. 3120)</td>
</tr>
<tr>
<td>Contractual Delivery Date</td>
<td>September 30, 2012</td>
</tr>
<tr>
<td>Actual Delivery Date</td>
<td>October 15, 2012</td>
</tr>
<tr>
<td>Contributing WPs</td>
<td>WP 9</td>
</tr>
<tr>
<td>Editor(s)</td>
<td>Christoph Sprenger (ETH Zurich)</td>
</tr>
<tr>
<td>Author(s)</td>
<td>Tigran Avanesov (INRIA)</td>
</tr>
<tr>
<td></td>
<td>Marianne Busch (LMU)</td>
</tr>
<tr>
<td></td>
<td>Juan Caballero (IMDEA)</td>
</tr>
<tr>
<td></td>
<td>Manuel Clavel (IMDEA)</td>
</tr>
<tr>
<td></td>
<td>Jorge Cuellar (SIEMENS)</td>
</tr>
<tr>
<td></td>
<td>Frédéric Dadeau (INRIA)</td>
</tr>
<tr>
<td></td>
<td>Carolina Dania (IMDEA)</td>
</tr>
<tr>
<td></td>
<td>Simone Frau (ETH Zurich)</td>
</tr>
<tr>
<td></td>
<td>Thai Son Hoang (ETH Zurich)</td>
</tr>
<tr>
<td></td>
<td>Aram Hovsepyan (KUL)</td>
</tr>
<tr>
<td></td>
<td>Felix Klaedtke (ETH Zurich)</td>
</tr>
<tr>
<td></td>
<td>Nora Koch (LMU)</td>
</tr>
<tr>
<td></td>
<td>Aliaksandr Lazouski (CNR)</td>
</tr>
<tr>
<td></td>
<td>Francesca Lonetti (CNR)</td>
</tr>
<tr>
<td></td>
<td>Eda Marchetti (CNR)</td>
</tr>
<tr>
<td></td>
<td>Paolo Mori (CNR)</td>
</tr>
<tr>
<td></td>
<td>Charles Morisset (CNR)</td>
</tr>
<tr>
<td></td>
<td>Martin Ochoa (SIEMENS)</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>Martin Wirsing (LMU)</td>
</tr>
<tr>
<td></td>
<td>Artsiom Yautsiukhin (CNR)</td>
</tr>
<tr>
<td></td>
<td>Koen Yskout (KUL)</td>
</tr>
<tr>
<td>Reviewer(s)</td>
<td>Juan Caballero (IMDEA)</td>
</tr>
<tr>
<td></td>
<td>Mass Lund (SINTEF)</td>
</tr>
<tr>
<td></td>
<td>Marinella Petrocchi (CNR)</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). Our research is divided into two main sub-domains: early assurance at the level of requirements, architecture and design using techniques such as refinement and model checking and complementary implementation-based assurance techniques such as testing and runtime verification.

In this deliverable, we report on further development of our methods and on initial prototypal support for security assurance for services that we have obtained during the second year of the NESSoS project. The work done so far in WP 9 covers the large majority of the tasks and activities set out in the NESSoS Description of Work. We record very good overall progress on all of the topics addressed, both in terms of the methods and of the tools developed. We can even report major advancements of the state-of-the-art in several areas, in particular in algorithmic verification, runtime monitoring, testing, and debugging. The large majority of methods and techniques are accompanied by a prototypal tool implementation or an extension of an existing tool.

Keyword List

Service assurance, software verification, refinement, model checking, testing, penetration testing, model-based testing, debugging, run-time verification, usage control, 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</td>
<td>Christoph Sprenger (ETH)</td>
</tr>
<tr>
<td>0.2</td>
<td>Draft of stable availability section</td>
<td>Marianne Busch (LMU)</td>
</tr>
<tr>
<td>0.3</td>
<td>Section on vulnerability prediction</td>
<td>Riccardo Scandariato (KUL)</td>
</tr>
<tr>
<td>0.4</td>
<td>Section on dangling pointers</td>
<td>Juan Caballero (IMDEA)</td>
</tr>
<tr>
<td>0.5</td>
<td>Section on quantitative access control</td>
<td>Charles Morisset (CNR)</td>
</tr>
<tr>
<td>0.6</td>
<td>Section on transformation from UWE to XACML</td>
<td>Nora Koch (LMU)</td>
</tr>
<tr>
<td>0.7</td>
<td>Section on runtime enforcement of XACML policies</td>
<td>Aliaksandr Lazouski (CNR)</td>
</tr>
<tr>
<td>0.8</td>
<td>Section on system compliance and policy enforcement</td>
<td>Felix Klaedtke (ETH Zurich)</td>
</tr>
<tr>
<td>0.9</td>
<td>Section on non-interference refinement in Rodin/Event-B</td>
<td>Son Thai Hoang (ETH Zurich)</td>
</tr>
<tr>
<td>0.10</td>
<td>Section on bridging model-based and language-based security</td>
<td>Jorge Cuellar (Siemens)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Martin Ochoa (Siemens)</td>
</tr>
<tr>
<td>0.11</td>
<td>Section on specification and verification of communicating authorization policies</td>
<td>Mohammad Torabi Dashti (ETH Zurich)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Simone Frau (ETH Zurich)</td>
</tr>
<tr>
<td>0.12</td>
<td>Section on analyzing ActionGUI models</td>
<td>Manuel Clavel (IMDEA)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Carolina Dania (IMDEA)</td>
</tr>
<tr>
<td>1.0</td>
<td>Complete draft</td>
<td>see page 3</td>
</tr>
<tr>
<td>2.0</td>
<td>Under internal review</td>
<td>see page 3</td>
</tr>
<tr>
<td>2.1</td>
<td>Revised version</td>
<td>see page 3</td>
</tr>
<tr>
<td>3.0.1</td>
<td>Approved by internal reviewers</td>
<td>Christoph Sprenger (ETH Zurich)</td>
</tr>
</tbody>
</table>
# Table of Contents

### LIST OF FIGURES ................................................................................. 9

### LIST OF TABLES.................................................................................. 11

### 1 INTRODUCTION ............................................................................. 13

1.1 Overview of 2nd-year results ............................................................ 13
1.2 Assurance along the NESSoS SDLC .................................................... 15
1.3 Document structure and NESSoS tasks .............................................. 16

### 2 EARLY ASSURANCE ..................................................................... 17

2.1 Stepwise refinement .......................................................................... 17
  2.1.1 Refining key establishment in Isabelle/HOL ........................................ 17
  2.1.2 Non-interference refinement based on Rodin/Event-B ......................... 19

2.2 Mapping security-design models to enable formal analysis .................. 20
  2.2.1 Analyzing ActionGUI models .......................................................... 20
  2.2.2 Transformation from UWE to XACML .............................................. 23

2.3 Algorithmic verification ..................................................................... 25
  2.3.1 Extension of CI-Atse with negative constraints handling .................... 25
  2.3.2 Specification and verification of communicating authorization policies ....... 26
  2.3.3 Stable availability under denial-of-service attacks through formal patterns .. 28

### 3 ASSURANCE IN IMPLEMENTATION ........................................... 29

3.1 Testing and debugging ....................................................................... 29
  3.1.1 jMuHLPSL cryptographic protocol mutation tool .................................. 29
  3.1.2 Testing authorization systems ......................................................... 31
  3.1.3 Early detection of dangling pointers ............................................... 34

3.2 Runtime verification ......................................................................... 36
  3.2.1 Compliance checking with incomplete knowledge .............................. 37
  3.2.2 On the enforceability of security policies ......................................... 38
  3.2.3 Runtime enforcement of XACML policies ........................................ 39

### 4 TRANSVERSE METHODOLOGY ................................................. 43

4.1 Bridging model-based and language-based security ............................ 43
  4.1.1 Non-interference on UML state-charts ............................................. 43

### 5 QUANTITATIVE METHODS .......................................................... 47

5.1 Predicting vulnerabilities in Android applications ............................... 47
  5.1.1 Application selection ..................................................................... 47
  5.1.2 Vulnerability prediction using code metrics ...................................... 47
  5.1.3 Vulnerability prediction using text analysis techniques ....................... 48

5.2 Quantitative access control ............................................................... 51
  5.2.1 Access Control Markov Decision Process ......................................... 51
  5.2.2 Access Control Partially Observable Markov Decision Process ............ 52
  5.2.3 Implementation with GLPK/GMPL ............................................... 53
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Relations to Other Work Packages</td>
<td>55</td>
</tr>
<tr>
<td>7 Conclusions</td>
<td>57</td>
</tr>
<tr>
<td>8 NESSoS WP 9 Second-year Publications</td>
<td>59</td>
</tr>
<tr>
<td>Bibliography</td>
<td>61</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1.1: Assurance techniques and development phases .................................................. 16
Figure 2.1: Refinement graph for key establishment protocols ............................................. 18
Figure 2.2: The Shadow operational semantics for sequential programs ............................... 20
Figure 2.3: The window Create_New_Professional .......................................................... 22
Figure 2.4: Toolchain for the model-driven approach ......................................................... 24
Figure 2.5: Model transformations to generate XACML policies ......................................... 24
Figure 2.6: Two processes (i.e. policy decision points), each with two threads ...................... 27
Figure 3.1: Mutation-Based Verification and Testing of HLPSL Protocols ......................... 29
Figure 3.2: Graphical User Interface of the jMuHLPSL Mutant Generator ......................... 30
Figure 3.3: Results of Mutant Filtering ............................................................................. 31
Figure 3.4: Authorization System Architecture .................................................................... 32
Figure 3.5: Number of use-after-free (left) and double-free (right) vulnerabilities in the CVE database in the 2008-2011 period, split by vulnerabilities in browsers, OSes, and other programs. 34
Figure 3.6: U-XACML Policy Enforcement Architecture .................................................... 39
Figure 3.7: Sequence Diagram of Usage Control Enforcement ............................................ 41
Figure 5.1: Prediction performance ..................................................................................... 49
Figure 5.2: Hello World Java file ....................................................................................... 49
Figure 5.3: Overall precision SVM ..................................................................................... 50
Figure 5.4: Overall recall SVM ......................................................................................... 50
List of Tables

Table 1.1: Mapping tasks to sections of this document ......................................... 16

Table 2.1: Refinement levels ................................................................ 17

Table 3.1: Experimental DATA ............................................................. 33

Table 5.1: Applications .................................................................... 48
Table 5.2: Reward functions ................................................................. 54
Table 5.3: Decision values for $\beta = 0.9$ and emergency probability of $p_e = 0.1$ ...................... 54
1 Introduction

The Future Internet is characterized by its highly dynamic and distributed nature. There is a strong tendency to rapidly and dynamically construct new applications and services from a given set of services that is already available. These services may be spread over many locations and involve a large number of stakeholders that do not necessarily trust each other. It is therefore natural that this evolution is paralleled by increasingly stronger security requirements on these applications. These are formulated in the form of security policies, the enforcement of which makes sure that, for example, services, agents, and other entities are properly authenticated and that sensitive data is only accessible to authorized players.

While the assurance of correctness and security properties is already a challenging task for centralized systems, this challenge is aggravated by the highly adaptive and rapidly evolving distributed systems and applications that populate the current and Future Internet. It is therefore indispensable that a suitable collection of rigorous and mathematically founded methods is developed that support the developer at each stage of the software development life cycle (SDLC). This goal is reflected in the task composition of Work Package 9 (WP 9) on Security Assurance for Services.

Rigorous development and verification techniques such as stepwise refinement, theorem proving, and model checking are adopted at the early phases of the SDLC, where they are most cost-effective (Task 9.1). Testing, debugging, and run-time verification methods (Task 9.2) are used in the later phases of the SDLC, complementing the previously mentioned techniques. Transversal methods (Task 9.3) aim to provide assurance support throughout the development process and to improve the integration and communication between the different phases. Finally, security and assurance metrics (Task 9.4) measure the overall security of the software-based services in order to provide developers with objective feedback on the achieved level of security.

This deliverable reports on the results obtained in WP 9 during the second 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 [116] and the initial results obtained during the first-year as reported in Deliverable D9.2 [115]. In the remainder of this section, we give an overview of our results, position the different methods and tools with respect to the SDLC, and relate the structure of this document to the WP 9 tasks.

1.1 Overview of 2nd-year results

Here, we summarize our the results from the second year. It is noteworthy that almost all methods and techniques we describe are supported by a prototypal tool implementation.

Stepwise refinement (Section 2.1) In stepwise refinement, we have worked along two axes, one based on classical refinement and the other on non-interference refinement.

First, we have extended our method for the development of security protocols by classical stepwise refinement to handle a larger class of protocols and we applied it to a substantial case study on key establishment protocols. The method as well as all models and proofs of the case study are implemented in the Isabelle/HOL theorem prover.

Second, we have investigated the possibility of using Event-B as a target language for translating ignorance-sensitive sequential programs, and used the Rodin Platform as a back-end to generate and discharge the required proof obligations for non-interference refinement. We have also proposed extensions of Event-B models with declarations to denote the visibility of variables with respect to sets of agents in order to enable a more practical embedding of ignorance-sensitive systems to Rodin/Event-B.

Mapping security-design models to enable formal analysis (Section 2.2) We can report on two lines of work in this area.

First, ActionGUI is a domain-specific language for modeling data-centric applications with fine grained access control policies. ActionGUI models are formal objects and therefore they can be reasoned about. We propose a formal methodology for proving two interesting properties of ActionGUI models, namely, that the sequence of actions triggered by the GUI events are invariant-preserving, with
with respect to the underlying data model, and that they are security-aware, with respect to the underlying security model. Our methodology is based on the formalization in OCL of these two properties. Then, to prove that the corresponding OCL expressions will hold in any possible scenario, we translate them into first-order logic and use SMT solvers to prove their logical properties.

Second, we translate access control policies specified in a high-level, graphical, modeling language (called UWE) to XACML policies. Thus, we make the specification of access control policies accessible to people not necessarily familiar with low-level languages such as XACML and we pave the way for formal analysis and development of such policies.

Algorithmic verification (Section 2.3) Here, we can report on progress along three lines.

First, we have extended the CL-Atse protocol verifier to handle negative constraints and applied it to the synthesis of secure mediators for service composition. The negative constraints increase the flexibility of the mediator synthesis. For instance, a negative constraint can specify that the mediator does not learn a certain message that is reserved to the eyes of trusted parties.

Second, we propose a formal language for specifying declarative distributed authorization policies that communicate through insecure asynchronous media. The interface between policy decisions and communication events is specified using guards and policy updates. We give trace semantics to communicating authorization policies, formulate a generic reachability problem, and show that this problem is decidable for a large class of practically-relevant policies. We have implemented our method in a tool.

Third, we propose a formal pattern-based approach to study defense mechanisms against DoS attacks on the availability of Internet services. We introduce the notion of stable availability, which means that with very high probability service quality remains very close to a threshold, regardless of how bad the DoS attack can get. We show by statistical model checking with the PVeStA tool that the composition of two known patterns yields a new improved pattern that guarantees stable availability at a reasonable cost while the individual patterns do not provide this guarantee.

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

First, we propose jMuHLPSL, a mutant generation tool that takes as input an HLPSL file formalizing a security protocol. The mutants are produced by applying mutation operators that represent common errors or implementation choices that can be made during the coding of the security protocol. These mutations may introduce security flaws, for which attack traces can be computed and reused as test cases for actual implementations of the protocol.

Second, we have worked on the testing of the implementation of the PolPA authorization system and in particular its Policy Decision Point (PDP), which determines whether access requests are allowed or denied. Based on the PolPA policy specification, we define a fault model and a test strategy able to detect the problems, vulnerabilities, and faults that may occur in a PDP implementation as well as a testing framework for the automatic generation of a test suite that covers the fault model.

Third, we propose early detection, a novel runtime approach for finding and diagnosing use-after-free and double-free vulnerabilities. While previous work focuses on the creation of the vulnerability (i.e., the use of a dangling pointer), early detection shifts the focus to the creation of the dangling pointer(s) at the root of the vulnerability. We implement our early detection technique in a tool called Undangle and evaluate Undangle for testing and vulnerability analysis.

Runtime verification (Section 3.2) In our work on system compliance and policy enforcement, we have focused on two aspects. First, we provide a solution to the problem of checking policy compliance in the presence of logging failures and disagreements between logged events. Our solution comprises a policy language and a monitoring algorithm. Second, we extend Schneider’s work on policy enforcement based on execution monitoring. We distinguish between controllable and observable system actions when monitoring executions and we give conditions for policy enforcement based on execution monitoring that are necessary and also sufficient.

Furthermore, we consider the problem of the run-time enforcement of usage control. We propose an enhanced authorization framework that deals with long-lasting accesses and it is able to interrupt
ongoing accesses when the corresponding access rights do not hold any more. We revise the OASIS XACML standard on addressing usage control features. We provide a policy schema, an architecture and an implementation of the framework based on the revised XACML.

**Bridging model-based and language-based security (Section 4.1)** We propose a methodology for automatically verifying the interaction of objects whose behavior is described by deterministic UML state-charts with respect to information flow policies, based on the so-called unwinding theorem. We have extended this theorem to cope with the particularities of state-charts: the use of variables, guards, actions and hierarchical states and derived results about its compositionality. In order to validate our approach, we report on an implementation of our enhanced unwinding techniques and applications to scenarios from the Smart Metering domain.

**Quantitative methods (Section 5)** We have devised two different techniques to predict which components of a software system contain security vulnerabilities. In the first approach, we have leveraged upon source code metrics to build a vulnerability prediction model with high precision, but low recall. In the second approach, we have exploited the potential of the bag-of-words representation and discovered that a dependable prediction model can be built by means of machine learning techniques. In a validation with 10 Android applications we have obtained performance results that often outclass state-of-the-art approaches.

Moreover, we have addressed the problem of quantitative access control by modeling an access control mechanism as a Markov Decision Process, extended this notion to Partially Observable Markov Decision Processes, and implemented our approach using linear programming through an example loosely inspired by the healthcare environment. An important aspect of our approach is to provide a metric for the value of security decisions, which is calculated as the potential impact of this decision. Hence, a security decision is no longer simply seen as a qualitative element, obtained from a static policy, but as a quantitative value computed from the environment, including several levels of uncertainty.

In Section 6 we discuss relationships to other NESSoS work packages. We draw conclusions and provide an outlook on future work in Section 7. Finally, in Section 8, we list the publications produced in this work package during the second year of the project.

### 1.2 Assurance along the NESSoS SDLC

The contributed assurance methods and tools described above are associated with different stages of the NESSoS SDLC as illustrated in Figure 1.1. The NESSoS SDLC is described in detail in Appendix A of Deliverable D10.2 [114] and summarized in Deliverable D10.3 [119]. The NESSoS SDLC is iterative, but we do not represent this fact in Figure 1.1 for simplicity.

Algorithmic verification is (in our context) associated with the design and analysis phase, testing and debugging corresponds to the SDLC stage following implementation, and runtime verification is mainly relevant for deployed systems. These techniques are complementary because they relate to different SDLC phases and thus different abstraction levels. The application of each of these techniques has the potential of strictly increasing the degree of security assurance obtained for a system under development.

Three methods have a transversal nature. First, model-based testing is a transversal form of testing, which relates an implementation to more abstract models that were produced in earlier SDLC phases. Second, stepwise refinement spans several SDLC phases from requirements to implementation. Refinement integrates development and assurance into a single activity that produces a series of models resulting in a system that is correct by construction. Third, quantitative methods may be used to evaluate the security of artifacts from any SDLC phase. These methods are closely related to risk and cost analysis, which is an integral stage of the NESSoS SDLC (succeeding requirements and preceding analysis and design, not show in Figure 1.1). In addition, we address how to bridge the gap between model-based security (at the design level) and language-based security (at the implementation level).
1.3 Document structure and NESSoS tasks

The mapping of NESSoS tasks listed in the Description of Work to the sections of this document is given in Table 1.1.

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
<th>Deliverable Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td><strong>Early assurance</strong></td>
<td>2</td>
</tr>
<tr>
<td>9.1.1</td>
<td>Stepwise refinement</td>
<td>2.1</td>
</tr>
<tr>
<td>9.1.2</td>
<td>Mapping security-design models to enable formal analysis</td>
<td>2.2</td>
</tr>
<tr>
<td>9.1.3</td>
<td>Extended model checking</td>
<td>2.3</td>
</tr>
<tr>
<td>9.2</td>
<td><strong>Assurance in implementation</strong></td>
<td>3</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Secure programming</td>
<td>(see D8.3)</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Testing and debugging</td>
<td>3.1</td>
</tr>
<tr>
<td>9.2.3</td>
<td>Run-time verification</td>
<td>3.2</td>
</tr>
<tr>
<td>9.3</td>
<td><strong>Transversal methodology</strong></td>
<td>4 (and 3 partially)</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Model-based testing</td>
<td>3.1</td>
</tr>
<tr>
<td>9.3.2</td>
<td>Bridging model-based and language-based security</td>
<td>4.1</td>
</tr>
<tr>
<td>9.4</td>
<td><strong>Security metrics</strong></td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1.1: Mapping tasks to sections of this document

Note that

- We have grouped our work in “Model-based testing” (Task 9.3.1) together with other work on testing and debugging in Section 3.1.
- Task 9.2.1 on “Secure programming” is not covered in this deliverable. However, deliverable D8.3 reports on work in secure programming involving assurance aspects (see also Section 6).
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. Section 2.1 covers stepwise refinement, where the focus is on constructing provably secure systems in a stepwise manner starting from requirements. In Section 2.2, we present formal mappings from security design models to other formalisms for which automated or semi-automated analysis tools are available. Finally, Section 2.3 is devoted to algorithmic verification techniques, which we extend for the increasingly expressive specification languages required for the modeling of Future Internet protocols and services, and their security properties.

2.1 Stepwise refinement

2.1.1 Refining key establishment in Isabelle/HOL

The fact that the development of even simple security protocols is error prone motivates the use of formal methods to ensure their security. The past decade has witnessed significant progress in post-hoc verification methods for protocol security based on model checking and theorem proving such as [12, 21, 43, 45]. However, methods for developing security protocols lag behind and protocol design remains more an art than a science.

In our view, a development method should be systematic and hierarchical, meaning that the development is decomposed into smaller steps that are easy to understand. These steps should span well-defined abstraction levels leading the developer from the requirements down to cryptographic protocols. The resulting protocols should be secure in well-established attacker models and such claims should ideally be supported by machine-checked formal proofs. Stepwise refinement provides such a hierarchical development method. However, most existing refinement-based approaches to developing security protocols [23, 28, 34, 38, 49, 88] fall short of at least one of these desiderata.

We have recently proposed a method for developing security protocols by stepwise refinement so that they are correct by construction [112]. The method consists of a four-level refinement strategy, summarized in Table 2.1, which allows the developer to build models that incrementally incorporate the system requirements and environment assumptions. Each model constitutes an idealized functionality for subsequent refinements. Safety properties, once proved for a model, are preserved by further refinements. These include (reachability-based) secrecy and authentication. In that work, we embedded this stepwise refinement method in the theorem prover Isabelle/HOL and used it to derive some basic unilateral entity authentication protocols and a simplified Otway-Rees key transport protocol without key confirmation.

<table>
<thead>
<tr>
<th>level</th>
<th>name</th>
<th>features</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>security properties</td>
<td>global, protocol-independent</td>
</tr>
<tr>
<td>L1</td>
<td>guard protocols</td>
<td>roles, local store, no channels, no messages</td>
</tr>
<tr>
<td>L2</td>
<td>channel protocols</td>
<td>channels with security properties, intruder</td>
</tr>
<tr>
<td>L3</td>
<td>crypto protocols</td>
<td>cryptography, Dolev-Yao intruder</td>
</tr>
</tbody>
</table>

Table 2.1: Refinement levels

In [113], we show how to systematically develop an entire family of key transport protocols. This family consists of the Needham-Schroeder Shared-key protocol (NSSK), the core of Kerberos 4 and Kerberos 5, and the Denning-Sacco protocol. Compared to the protocols developed in [112], these protocols are significantly more complex in size and message structure and exhibit a number of additional features and security properties such as the use of timestamps, replay protection caches, encrypted tickets (double encryption), dynamically created communication channels, key confirmation, key freshness, and key recentness.

Figure 2.1 displays the graph of refinements of our development. We start our development from abstract, protocol-independent models of secrecy (s0) and authentication (a0i and a0n). Our refinements
proceed through the abstract protocol levels L1 and L2 until we obtain the desired concrete cryptographic protocol that is secure against a standard Dolev-Yao intruder. Each secrecy or authentication property is established by refining an appropriate instantiation of the respective L0 model.

A central and novel feature of our approach is the use of guard protocols (L1) as an intermediate abstraction linking security properties (L0) and message-based protocols (L2-3). Guard protocols enable the straightforward abstract realization of security goals by adding security guards as necessary conditions for the execution of certain protocol steps. Different kinds of security guards ensure the preservation of different properties such as secrecy, authentication, and recentness. For example, key secrecy means that only authorized agents may know a key. Accordingly, steps of guard protocols where an agent $A$ learns a key $K$ contain a guard requiring that $A$ is authorized to know $K$. For authentication, there are guards ensuring that the local state of an agent (partially) agrees with the state of another agent.

The security guards for secrecy and authentication communicate with other agents by accessing their local stores. This abstraction simplifies proofs, but is not directly implementable in a distributed setting. Hence, we implement these guards at Level 2 by receiving messages on channels with intrinsic security properties. The associated refinement proof naturally gives rise to invariants stating that the receiving of channel messages implies the security guards they implement. These invariants precisely state the security properties guaranteed by the messages. For example, a message containing a key $K$ received on a confidential channel to agent $A$ may implement a guard authorizing $A$ to learn $K$. The corresponding invariant guarantees that $A$ is authorized to learn $K$ from this message.

**Contributions.** Our contributions are threefold. First, we show how to develop an entire family of key transport protocols from requirements down to cryptographic protocols that are secure against a standard Dolev-Yao intruder. We model realistic features that are often abstracted away, such as replay prevention caches for timestamped messages to achieve strong properties like injective authentication. We have formalized all models and proofs in [113] in the Isabelle/HOL theorem prover. Our formalization includes a new infrastructure with general Isabelle/HOL theories for protocol runs, fresh values, and channels with security properties. This supersedes the protocol-specific embeddings of these concepts in [112].

Second, our development provides evidence that guard protocols constitute a fundamental abstraction that bridges the gap between security properties and standard message-based protocol descriptions. In other approaches, the guarantees about protocol messages given by the invariants mentioned above

---

Our entire development including the infrastructure theories is available at [http://people.inf.ethz.ch/csprenge/csf12](http://people.inf.ethz.ch/csprenge/csf12).
and the associated reasoning are usually stated informally (if at all). By formalizing them, our approach fosters clear protocol designs and simple abstract security proofs. Moreover, in message-based protocol descriptions, secrecy and authentication are often not clearly separated (e.g., when using a secure channel providing both properties) or are interdependent (e.g., due to a layered use of cryptographic keys and operations). This is a major source of complexity and errors and makes security protocols hard to design and understand. In contrast, guard protocols realize secrecy and authentication properties abstractly, independently, and in a straightforward way. These features of guard protocols facilitate the formal development of secure protocols and underscore the central role they can play in property-driven design approaches.

Third, our development shows that our method scales to protocols of realistic complexity. We only need one extension: to model key confirmation at Level 2, we have added dynamically created channels to our modeling infrastructure. The protocols in our development share both structure and properties as the graph of refinements of our development indicates (see Figure 2.1). Property preservation through refinements avoids proof duplication and fosters well-structured proofs.

For further details on this work, we refer the reader to our publication [113].

### 2.1.2 Non-interference refinement based on Rodin/Event-B

The “Shadow Semantics” is a qualitative model for noninterference security introduced by Morgan [98]. The framework is designed for reasoning about noninterference-style properties for sequential programs. With the adoption of the traditional noninterference separation into high- and low-security variables, an adversary within the Shadow framework is governed by the assumption that he/she may observe directly the low-security (visible) variables but can only infer/guess the values of the high-security (hidden) variables from these observations. The framework includes an assertion language for expressing “knowledge”/“ignorance” together with a weakest-precondition modal semantics, which can be used as the basis for ignorance-preserving refinement. An attractive property of this work is the possibility to translate (1) ignorance-sensitive programs into traditional ones and (2) the properties into first-order logic (the “shadow form”), and to reason about (1) and (2) using traditional methods. Examples of protocols formally developed from specifications to implementations using the Shadow Semantics include the Oblivious Transfer [106], the Dining Cryptographers [39].

Assume that our program state is partitioned into a “visible” part $v$ and a “hidden” part $h$ and our program operates over $v$ and $h$. Here the properties that we are interested in are what information an observer knows about the part of the program states that he cannot directly see, i.e. $h$. In other words, we can ask the question “from the final value of $v$, what can an observer deduce about the final value of $h$” [98]. The answer obviously depends on the actual program: If the program is $v ::= 0$ then what the observer knows is just the same as what he knows before executing the program; if the program is $v ::= h$ then he knows the exact value of $h$; if the program is $v ::= h \mod 2$, then he knows the parity of $h$ (in addition to what he already knows about $h$).

Assume a state space with only two sets of variables: visible variables $v$ and hidden variables $h$, an additional variable $H$ – called the shadow of $h$ – which keeps all the values that $h$ has potentially at any point. It is required that $h \in H$. The Shadow operational model is given by translating the $(v, h)$-programs (ignorance-sensitive) to traditional $(v, h, H)$-programs (the shadow form) as shown in Figure 2.2. In this figure, $[S]$ denotes the translation of $(v, h)$-program $S$ into a traditional $(v, h, H)$-program and variable $e$ is fresh. The sequential language contains deterministic assignments ($\equiv$), non-deterministic assignments ($\subseteq$), demonic choices ($\hat{\equiv}$), sequential compositions ($\&$), and conditional statements (if ... then ... else ... end).

Given two program statements $S$ and $T$, we say $S$ is refined by $T$ (denoted as $S \sqsubseteq T$) when, starting from some before state $(v, h, H)$, every possible after state $(v', h', H')$ of $T$ can be matched by an after state $(v', h', H'_S)$ of $S$, where $H'_S \subseteq H'_T$. Intuitively, shadow refinement corresponds to standard functional refinement on traditional variables $v$ and $h$, with the possibility of enlarging the shadow $H$ component (shadow refinement).

Properties of ignorance-sensitive programs are modeled using an assertion logic for expressing knowledge from Morgan [98, 99]. Informally, the logic is defined to be first-order logic augmented with a modal operator “know” $K$ [50]. $K\phi$ (read “know $\phi$”) holds in a state when $\phi$ holds in every (other) state “compatible” with the visible part of this state, the program text and the information about execution path as well as earlier visible values. The dual operator of $K$ is $P$ (hence $P\phi$ read “possibly $\phi$”) is defined as $P\phi \equiv \neg K(\neg\phi)$. 
\[ \begin{align*}
  v &:= E(v, h) \\
  [v] &:= E(v, h) \\
  [v : E(v, h)] &:= E(v, h) \\
  [v : E(v, h)] &:= E(v, h) \\
  h &:= E(v, h) \\
  [h] &:= E(v, h) \\
  [h : E(v, h)] &:= E(v, h) \\
  [S \cap T] &:= S \cap T \\
  [S; T] &:= S; [T] \\
  \text{if } G(v, h) \text{ then } H &:= \{h \in H \mid G(v, h)\}; [S] \\
  \text{else } H &:= \{h \in H \mid \neg G(v, h)\}; [T] \\
  \text{end}
\end{align*} \]

Figure 2.2: The Shadow operational semantics for sequential programs

Contributions. In [79], we investigated the possibility of using Event-B [5] as a target language for translating ignorance-sensitive sequential programs, and used the Rodin Platform [6] as a back-end to generate and discharge the required proof obligations for shadow refinement. The sequential programs contain several labeled assignment statements. The tool translates these input programs (together with some declarations about variables, function constants) into Event-B models. The initial version of the tool translates each statement into Event-B events with an additional assignment to the shadow variable \( H \). Sequential steps are modeled in Event-B using appropriate control flags. In order to accommodate the proof of some refinements, a later version of the tool translates each assignment statement into two events: one corresponding to the functional update to the ordinary variables \( v \) and \( h \), one corresponding to the shadow update to the shadow variable \( H \). Control variables are added accordingly to schedule the functional update and shadow update events.

The work done in [79] only focuses on ignorance-sensitive sequential programs: the expressiveness of Event-B for more general transition systems was not explored. Furthermore, splitting the functional and shadow update, corresponding to an assignment statement, complicates the formal model (somewhat artificially). In [78], we give some remedy for the above problems by considering the shadow semantics within transition systems of Event-B. Our proposal is to extend Event-B models with declarations to denote the visibility of each variable with respect to some set of agents. Moreover, additional knowledge properties (i.e., expressed using modal operators \( K \) and \( P \)) of the models can be stated as invariants of the models. From an ignorance-sensitive Event-B model, we generate different shadow Event-B models, depending on the agent’s point-of-view. Each shadow Event-B models contains additional shadow variable \( H \). The practicality of approach includes identifying two patterns of modal invariants that are correct-by-construction. These invariants are important for proving the shadow refinement between different abstract levels.

2.2 Mapping security-design models to enable formal analysis

2.2.1 Analyzing ActionGUI models

In this section we report on our initial methodology for supporting formal analysis of ActionGUI models. Within NESSoS, ActionGUI is being used in the EHealth Case Study proposed in Workpackage 11 (see Deliverable 11.3 [118]). We intend then to apply this methodology to formally analyze the ActionGUI models that are being proposed in the design phase of the development of the EHealth Records Application.

ActionGUI is a domain-specific language for modeling data-centric applications with fine-grained access control policies. It supports full model-driven engineering. This means, in particular, that ready to be deployed data-centric, secure web applications can be automatically generated from ActionGUI models. ActionGUI also supports the principle of separation of concerns. In particular, ActionGUI models consists
of three models:

- A data model that defines the application domain, and, therefore, all possible scenarios.
- A security model that defines the application’s fine-grained access control policy.
- A GUI model that defines the application’s graphical interface.

ActionGUI models are formal objects and therefore they can be reasoned about. Next, we first introduce a specific property about ActionGUI models, namely, invariants preservation. To illustrate this property, we use a real example borrowed from the models that we have developed in ActionGUI for the EHealth Case Study [118]. Then, we sketch our initial methodology to formally prove this property. Finally, we briefly discuss another interesting property, namely, security-awareness, that can be also proved using our methodology.

**ActionGUI models’ properties: invariants preservation.** In Figure 2.3 we show a screenshot of a window Create_New_Professional, where a user with the Director role can add a new professional to a database. Recall that, in our EHealth Case Study, professionals must satisfy, among others, the following invariants:

- Each professional has a unique username or login, different from the empty string.
- Each professional has exactly one role (Director, Doctor, Nurse, or Administrative).
- Each professional is assigned to exactly one Department.

Therefore, we can expect the ActionGUI model of the window Create_New_Professional be such that the following properties —both required for preserving the database’s invariants— hold:

- Any sequence of actions triggered when clicking upon the button Create_Professional will perform the appropriate changes in the database so as the aforementioned invariants are preserved. For example, they will always assign both a role and a Department to a newly created professional.
- The conditions for triggering any sequence of actions after clicking upon the button Create_Professional guarantee that these actions will always have the appropriate arguments so as the aforementioned invariants are preserved. For example, they will always check that both a role and a department have been actually chosen, i.e., that they are not “undefined”, before triggering any sequence of actions.

Next, we introduce more formally the invariants preservation property and sketch our initial methodology to formally prove this property.

**Checking ActionGUI models’ properties: invariants preservation** We can characterize invariants preservation as follows:

In any possible scenario, for any sequence of data actions triggered by a GUI event, if all the data model invariants are satisfied before the event, then each of them will be also satisfied after the event.

Notice that the actual sequence of actions triggered by an event will depend on the evaluation, at run-time, of the conditions constraining the execution of these actions. Notice also that, since we are currently not considering iterative actions (or loops), any sequence of actions triggered by an event will be finite. Hence, the possible sequences of actions triggered by an event $evt$ can be characterized by a set of (symbolic) traces of the form:

$$tr = \langle (c_1, a_1), \ldots, (c_m, a_m) \rangle,$$

where each pair $(c_i, a_i)$ specifies that, before executing the action $a_i$, the condition $c_i$ holds. In this setting, to prove that an event $evt$ is invariant-preserving, we must show that, for any scenario of the given data model, for any invariant $inv$ and for any trace $tr$ of the event $evt$, if every invariant of the data model holds in the given scenario, then the invariant $inv$ will also hold after executing (the corresponding) instance of the trace $tr$. According to our methodology, this can be proved as follows: Let $\Lambda$ be an ActionGUI model, and let $evt$ be an event in the GUI model of $\Lambda$. Then, to prove that $evt$ is invariant-preserving:
Figure 2.3: The window *Create_New_Professional*. 
Step 1: From the GUI model of \( A \) we deduce the finite set of traces \( TR \) of the event \( evt \), characterizing, as explained above, all the possible sequences of actions that this event can trigger. We have developed a prototype that provides automatic support for carrying out this step.

Step 2: For every invariant \( inv \) declared in the data model of \( A \), and any trace \( tr \in TR \) of the event \( evt \), we formalize in OCL that \( evt \) is invariant-preserving with respect to \( inv \) and \( tr \). We have developed a prototype that provides automatic support for carrying out this step.

Step 3 For every OCL expression \( exp \) generated in Step 2, we check its validity as follows:

- We transform \( exp \) into a first-order logic formula using OCL2FOL [42].
- We (try to) automatically prove that the negation of this formula is unsatisfiable using an SMT solver (e.g., Z3 [51] or Yices [58]).

Other ActionGUI models’ properties: security-awareness. Another interesting property of ActionGUI models is what we called security-awareness, that can be characterized as follows:

In any possible scenario, for any sequence of action triggered by a GUI event, if all the data model invariants are satisfied before the event, then, for any action in this sequence, its authorization constraint will always be satisfied.

This property is clearly interesting for improving the efficiency of the generated application: if an ActionGUI model is proved to be security-aware, then no authorization constraints need to be checked at run-time before executing a data action.

Security-awareness can also be proved using our methodology. Notice, however, that to prove that an event \( evt \) is security-aware we must show that, for any scenario of the given data model, for any role \( rl \) in the given security model, for any trace \( tr \) of the event \( evt \) and for any pair \( (c_i, a_i) \) in \( tr \), the authorization \( auth \) that constrains the execution of the action \( a_i \) for (users with) the role \( rl \) will hold after executing the subtrace \( \langle (c_1, a_1), \ldots, (c_{i-1}, a_{i-1}) \rangle \) of \( tr \). Here we can also assume that every invariant of the data model holds in the given scenario.

Contributions. We have presented a methodology to formally prove two properties about ActionGUI models: invariant preservation and security awareness. Within NESSoS, ActionGUI is being used in the EHealth Case Study proposed in Workpackage 11 (see Deliverable 11.3 [118]). To prove the aforementioned properties we reduce them to unsatisfiability problems in first-order logic. For this, we have formally defined in OCL the post-conditions corresponding to the different data-actions. We use OCL2FOL to map all the relevant OCL expressions into first-order logic, and SMT solvers to automatically resolve the unsatisfiability problems. This is still work in progress. Also, we are developing a tool to support this methodology, and we plan to apply it to formally analyze the ActionGUI models for the EHealth Case Study proposed in Workpackage 11 (see Deliverable 11.3 [118]).

2.2.2 Transformation from UWE to XACML

Access control is a fundamental means for restricting what operations (authenticated) users can perform on protected resources. Different access control systems have been developed to support security. The main components of these systems are the security policies, the security model, and the implementation mechanisms [50]. The policies define the rules according to which access control must be regulated. The model provides a formal representation of the policies and how they work. The mechanisms define how the controls imposed by the policies and the model are implemented.

Languages for access control aim at supporting the expression and enforcement of policies. Several such languages have been proposed. Many of them are XML-based, e.g., the OASIS standard eXtensible Access Markup Language (XACML) [101]. XACML defines an access control policy language providing features to express authorization rules in XML for resources also defined in XML. However, the XML syntax of XACML can make the task of writing policies difficult and error-prone, and it is not adequate for formally defining the semantics of the language and reasoning about them. Other languages rely
on concepts and techniques from logic, which instead offer the advantage of a formal foundation and a well-defined analysis, but have the drawback of not being usable for a wide spectrum of users.

Access control languages might thus be too low level and impractical for developers accustomed to work with abstract architectural models of systems.

This model-driven process\(^2\), shown in Figure 2.4, solves the problems mentioned above. It offers the advantages of an easy to learn and intuitive visual specification language for policies, which can be also translated automatically to a formal specification enabling evaluation of policies and access requests through the FACPL Policy Decision Point [32]. The central part of the tool chain is also integrated in the SDE (see Chapter 5.2 of [121]).

\(^2\)Instructions and software for installing the tool chain can be found at http://uwe.pst.ifi.lmu.de/uwe2facpl

Intuitively, the UWE2XACML transformation generates a \(<PolicySet>\) for each role, each of which contains one \(<Policy>\) for any class connected to the considered role. Furthermore, a single \(<Policy>\) is used to deny access to all resources not specified in the \(<PolicySet>\), which is the default behavior of UWE’s basic rights models (see Figure 2.5).

Notably, to allow a sub-role of a given role to use the permission specified by the super-role, the target of the \(<PolicySet>\) corresponding to the super-role is extended to also match requests from the sub-role.

Each \(<Policy>\) for a constrained class contains one \(<Rule>\) for each action between the role and the class. Attributes targeted by +\textit{All} actions are divided into a set of \(<Resources>\), omitting those from the except tag. OCL constraints inside UML comments with authorizationConstraint stereotype are transformed to a \(<Condition>\). The condition is located within a \(<Rule>\) representing the appropriate action. For the time being, we implemented only a few basic OCL constraints.

The transformation, performed by the XACML2FACPL component in Figure 2.4, is straightforward. Its flow loops over the policy sets creating the necessary data structures for the FACPL representation. The original XML document is read by using JAXB\(^3\). The loop over the elements is driven by the XACML schema definitions by traversing its data types.

\(^3\)JAXB. http://jaxb.java.net
Contributions. Our contribution published in [32] is twofold. On the one hand, we ease the specification of access control policies making it accessible to people not necessarily familiar with access control languages. On the other hand, we provide a formal specification of the policies that is generated automatically and enables further analysis and verification of these access control policies. The implementation of our approach consists of an initial specification of the security requirements using a high-level, graphical, modeling language (called UWE [84]), thus also providing a human understandable view of the policies in force at the system, and then to automate the policy development process towards the formally founded language (called FACPL [93]) by means of a suitable software tool chain. The tool chain comprises two transformations: UWE2XACML and XACML2FACPL.

For further details on this work we refer the reader to our publication [33].

2.3 Algorithmic verification

2.3.1 Extension of CL-Atse with negative constraints handling

Intruder constraints express the (minimum) set of deductions or operations that a Dolev-Yao intruder should perform to acquire knowledge and possibly interact with other agents to lead a protocol or a Web-Services session to some insecure state occurring in a given trace [11, 122]. By exploring all elementary traces, which are basically sequences of agent names such as "Alice plays, then Bob plays, then Alice again, etc.", the CL-Atse tool checks the virtually infinite set of possible attacks states by verifying whether the union of intruder constraints generated along these traces are compatible with the negation of some specified security properties (also expressed by constraints). If these constraints are satisfiable, the intruder will be capable of invalidating one of the security properties at some state. Until now, all intruder constraints considered in CI-Atse (and related tools) were limited to positive constraints. However we believe that negative intruder constraints give more flexibility to explore the set of possible attacks. For instance, with negative intruder constraints we can rule out certain attacks where the intruder would exploit some data (e.g. because (s)he would get detected for some external reason) and focus the search on alternative attacks.

Technical details. The implementation of negative intruder constraints in CI-Atse follows assumptions and ideas presented in [13]. In this paper, we present a non-deterministic decision procedure obtained by guessing and checking solutions of size polynomially bounded by inputs. The practical method implemented in CI-Atse rather operates by collecting and reducing sets of constraints but the idea is the same and the collected sets of constraints represents the same states as in [13]. The decidability result in [13] applies to the class of subterm convergent theories and therefore covers many interesting theories in formal security analysis. Negative constraints on the intruder’s knowledge are limited to contain variables already occurring in earlier positive constraints. The method does not apply to theories with algebraic operators like Xor and Exp, so the models presented to the tool should not include Xor or Exp operators and negative intruder constraints at the same time. Currently, no constraint solving technique is known to be correct and complete without these conditions. But within these hypotheses, the tool support is correct and complete for any Aslan specification.

The theoretical result cited above has implied too that negative constraints can be managed in a lazy way by the tool. That is, the tool collects negative constraints along with positive ones, and stores them as usual as an ordered sequence based on the trace. Positive constraints are reduced the usual way, down to elementary constraints. A standard approach would be to solve negative constraints using advanced and provably non-terminating reduction techniques, with term unifications and replacements. But this is highly impractical. Fortunately, the analysis in [13] shows that in fact we only need to apply instead basic syntactic deduction techniques to slightly reduce the negative constraints according to the choices done when reducing the positive ones. This is sufficient to guarantee satisfiability if the constraints are not "trivially" reduced to false. Therefore, while a negative constraint elimination method would have certainly refined the solutions expressed by the positive constraints, this is not needed since for satisfiability simple checks on negative constraints are sufficient. This is the approach followed by CI-Atse.

Negative constraints are not completely reduced at the end, that is, a solution found by CI-Atse might still contain variables, and thus represents in fact a set of solutions along with remaining (negative) con-
straints. This happens even with positive-only constraints in Cl-Atse, since variables may occur in the output too. This is not a problem with positive constraints, as variables in attack traces produced by the tool represent free values, which can be replaced by arbitrary ones. However, negative constraints provide less freedom, since the chosen values may not satisfy the remaining negative constraints. But, the above work gives an algorithm to fill the remaining free variables with fresh values in a careful way that preserves the remaining negative constraints. The algorithm is simple but it is almost useless in practice, and it will not build complex structured values for these variables. Therefore, Cl-Atse does not employ it and leaves the remaining variables uninstantiated in the attack, as for positive-only constraints.

Experiments. On the Cl-Atse website\(^4\), one can find a non-trivial application of negative intruder constraints and their implementation as above. This application, inspired by the Loan Origination case study from Avantssar, is an orchestration problem where the mediator has the power to respond to the client’s request directly but is obliged to delegate some parts of the process to external agents because some client data (e.g., an amount of a loan request) must remain unknown to the mediator and known only by trusted “Clerk” agents. In Avantssar framework we were not able to express directly non-disclosure policies nor separation-of-duty policies. In NESSoS we want to express and handle these important policies and this gets possible thanks to the introduction of negative constraints as follows. According to Avantssar’s orchestration method, this orchestration problem is encoded into Cl-Atse analysis back-end as a standard web-services insecurity problem in presence of an active intruder. Thus, any intruder playing an attack found by the tool can be compiled to a mediator, i.e., as a new agent representing a solution to the orchestration problem. Thus, a security policy like “the mediator should not know X” is encoded as “the intruder must not know X” in Cl-Atse, i.e., as a negative constraint on the intruder's knowledge. The need for these constraints can be easily shown by altering the model:

1. Without the negative constraints, Cl-Atse finds an orchestration where the mediator assumes himself to be a Clerk, i.e., this is precisely the behavior we want to avoid;
2. Without the clerks, Cl-Atse does not find any orchestration. This is logical since without any external agent playing a clerk, and with the policy preventing the orchestrator to be a clerk himself, it is impossible to satisfy the client;
3. Without removing anything, Cl-Atse finds a good orchestration where the mediator delegates some parts of the process to the Clerks, i.e. the secret data remains secret and non-deducible by the mediator.

Contributions. The protocol and Web services analysis tool Cl-Atse has been extended with the ability to solve intruder negative constraints. The main application of negative constraints we have explored so far is the synthesis of a secured mediator for the service composition.

For further details on this work we refer the reader to our publication [13].

2.3.2 Specification and verification of communicating authorization policies

Ideally, by enforcing distributed authorization policies, the behavior of a distributed system should be constrained so that the distributed system achieves its functional goals without ever entering an insecure state. For example, in a hospital, a typical functional goal is to enable the medical personnel to access the health records of their patients. Here, an instance of the unreachability of insecure states is: no one else can (ever) access these records. Decentralized trust management systems, or distributed authorization logics, play a pivotal role in securing distributed systems [4,22,30,53,71]. They allow us to formally reason about distributed authorization policies, even prior to their deployment.

We consider policy decision points (PDPs) that communicate with each other by exchanging messages over insecure media. The policies of such PDPs change due to receive events (e.g., upon receiving a public key certificate), constraining the communication events (e.g., access tokens are sent only to the principals whose credentials have not been revoked). As PDPs communicate over insecure media, attacks may occur: expired certificates are replayed, certificate revocation lists are delayed, messages

\(^4\)http://cassis.loria.fr/Cl-Atse
are tampered with, etc. We let the attacker be in direct control of the communication media. This view is motivated by the workings of security-sensitive distributed services, such as federated identity management systems (OAUTH [75], etc.). We define a formal language for specifying communicating authorization policies, and give an algorithm for deciding reachability for a large class of such policies. This builds upon previous work on analyzing security-sensitive distributed services [14,63] and the formalisms of [62,72].

In our formalism [64], we model communicating authorization policies as a finite number of processes. Intuitively, a process represents a PDP. Each process consists of a finite number of threads that share a policy. Threads are finite sequences of communication events; they run in parallel and exchange messages with the threads of other processes over insecure media. The policy of a process is a (declarative) program which models the shared authorization policy that the threads of the process evaluate.

Threads communicate by sending and receiving messages, as is common in asynchronous message passing environments. Each send event is constrained by a guard, and each receive event results in an update of the policy of a process. Intuitively, guards and updates belong to the policy level, as opposed to send and receive events, which constitute the communication level (Figure 2.6). From an operational point of view, guards are statements that, if derivable from the policy of a process, allow the process to perform a corresponding send action. Updates are also statements at the policy level. When a process receives a message in one of its threads, it updates its policy correspondingly. Intuitively, updates associate meanings to the messages a process receives in terms of statements at the policy level. For example, a signed X.509 certificate sent by a certificate authority means that the authority endorses the public key and its owner, mentioned in the certificate.

We assume an all-powerful attacker who is in direct control of the insecure communication media. In fact, the messages the PDPs exchange are passed through the attacker. This is a common (worst-case scenario) assumption in the literature. The message inference capabilities of the attacker may reflect the Dolev-Yao threat model [57]. For example, the attacker can indirectly manipulate the policies of the participating PDPs by sending tampered messages which affect the update statements.

**Contributions.** Our main contributions are: (1) We present a formalism for specifying communicating authorization policies, and their hostile environment. (2) We give an algorithm for deciding reachability of communicating authorization policies. Given a formal model of the policy decision points and the attacker, we can decide (under certain conditions, specified precisely in [65]) whether, or not, an (insecure) state is reachable. For instance, questions of the form “Can the attacker learn the content of a certain file?” or “Can Ann ever be authorized to access a certain document stored at the file server?” are expressible as reachability problems. Note that communication level events, policy level computations, and the interface between the two are all taken into account when deciding reachability. This sets apart our decision algorithm from those for deciding secrecy in security protocols, and those for deciding reachability in dynamic authorization logics (for a detailed comparison see [65]). The set of communicating authorization policies for which we can decide reachability is of practical relevance. We demonstrate this by formalizing an electronic health record system described in [118].

For further details on this work we refer the reader to our publication [64] and its extended version [65].
2.3.3 Stable availability under denial-of-service attacks through formal patterns

On December 8, 2010 at 07:53 AM EDT, MasterCard issued a statement that “MasterCard is experiencing heavy traffic on its external corporate website [. . .]. There is no impact whatsoever on our cardholders ability to use their cards for secure transactions” [94]. In fact, by that time, a distributed denial-of-service attack (DoS) brought the website down and made their web presence unavailable for most customers for several hours.

Availability is an important security property for Internet services and a key ingredient of most service level agreements. It can be compromised by distributed denial-of-service (DoS) attacks. DoS defense mechanisms help maintaining availability; nevertheless even when equipped with defense mechanisms, systems will typically show performance degradation. Thus, one of the goals of security measures is to achieve stable availability, which means that with very high probability service quality remains very close to a static constant quantity, regardless how bad the DoS attack gets. Cloud computing offers the possibility of dynamic resource allocation and thus can be used to leverage stable availability when combined with DoS defense mechanisms.

Contributions. In our work we propose a formal pattern-based approach to study defense mechanisms against DoS attacks. We enhance pattern descriptions with formal models that allow the designer to give guarantees on the behavior of the proposed solution. The underlying executable specification formalism we use is the rewriting logic language Maude [41] and its real-time and probabilistic extensions.

We introduce the notion of stable availability, which means that with very high probability service quality remains very close to a threshold, regardless of how bad the DoS attack can get. We use formal patterns which, in addition to “normal” patterns, come with formal guarantees and enable automated pattern composition, often resulting in semi-automatic construction of new models with improved properties. We use this pattern-based approach to study defense mechanism against DoS attacks in a model-based setting.

We present two formal patterns which can serve as defenses against DoS attacks:

- the Adaptive Selective Verification (ASV) pattern, which enhances a communication protocol with a defense mechanism, and
- the Server Replicator (SR) pattern, which provisions additional resources on demand.

However, ASV achieves availability without stability, and SR cannot achieve stable availability at a reasonable cost. We have analyzed properties of the ASV+SR pattern using the statistical model checker PV\textsuperscript{ES}T\textsuperscript{A}, and have shown that, unlike the two original patterns, ASV+SR achieves stable availability in presence of a large number of attackers at reasonable cost, which can be predictably controlled by the choice of the overloading parameter.

The ASV protocol is a well-known defense against DoS attacks in the typical situation that clients and attackers use a shared channel where neither the attacker nor the client have full control over the communication channel [82]. The ASV protocol adapts to increasingly severe DoS attacks and provides improved availability. However, it cannot provide stable availability. By replicating servers one can dynamically provision more resources to adapt to high demand situations and achieve stable availability; but the cost of provisioned servers drastically increases in a DoS attack situation. These two patterns are modeled in Maude and then formally composed to obtain the new improved ASV+SR pattern.

For further details on this work we refer the reader to our publications [59, 125].
3 Assurance in Implementation

Several assurance techniques are available to 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. Once a potential problem is found, we need efficient debugging techniques that help us to understand the cause of the fault or attack. Debugging techniques are also needed to detect memory management and other implementation-specific problems that occur due to the continued use of low-level programming languages without garbage collection. Such problems constitute some of the most dangerous attack vectors. Testing and debugging are 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 topic is discussed in Section 3.2.

3.1 Testing and debugging

3.1.1 jMuHLPSL cryptographic protocol mutation tool

Mutation testing has been introduced by DeMillo [52]. This technique consists in introducing a single modification (simulating a realistic fault) into a program, or a model. Such a technique can be used at two levels. When applied on the code of a program, it is used to evaluate a given test suite, in terms of fault detection capabilities. When applied on a model (our choice), it can be used to compute test cases, by creating tests that must be able to exhibit the considered fault.

Principles behind the jMuHLPSL tool. jMuHLPSL [48], which we have summarized in Deliverable D9.2 [115], is a mutant generator tool that takes as input a verified HLPSL protocol, and computes mutants of this protocol by applying systematic mutation operators on its contents. As depicted in Figure 3.1, the mutated protocol then has to be analyzed by a dedicated protocol analysis tool (here, the AVISPA tool-set [12]). Three verdicts may then arise. The protocol can still be safe, after the mutation, this means that the protocol is not sensitive to the realistic “fault” represented by the considered mutation. This information can be used to inform the protocol designers of the robustness of the protocol w.r.t. potential implementation choices, etc. The protocol can also become incoherent, meaning that the mutation introduced a functional failure that prevents the protocol from being executed entirely (one of the participants remains blocked in a given non-final state). The protocol can finally become unsafe when the mutation introduces a security flaw that can be exploited by an attacker. In this case, the AVISPA tool-set is able to compute an attack-trace, that represents a test case for the implementation of the protocol. If the attack can be replayed entirely, then the protocol is not safe. If the attack cannot be replayed then the implementation does not contain the error introduced in the original protocol.

Figure 3.1: Mutation-Based Verification and Testing of HLPSL Protocols
Mutation operators. We propose seven mutation operators, which represent an implementation choice made by the developer of the protocol, or an error that can be made when implementing the protocol. In both cases, mutating the protocol may introduce security flaws.

- Exclusive-OR: forces the encryption schemes to use the exclusive-or operator.
- Exponential: forces the encryption schemes to use the exponential operator.
- Homomorphism: simulates the homomorphism property of a given encryption scheme, by replacing the encryption of a concatenated parts of a message by the concatenation of the encrypted parts of the message.
- Public keys: forces several participants to share the same public key.
- Substitution: replaces a part of a message by an arbitrary constant, to simulate the uselessness of this message part.
- Hash functions: suppresses the use of hash functions inside the protocol.
- Permutation: swaps the blocks inside the messages.

These mutations are applied at the abstract level of the HLPSL protocol and are then analyzed by the AVISPA tool-set. Further details regarding how the mutation is applied, and why the mutation has been proposed, are available in the original paper [48].

Experiments. For our experiments, we have developed a GUI (shown in Figure 3.2) that makes it possible to select a given HLPSL protocol and choose the mutation operators to apply on it. We applied our mutation operators to a set of 50 safe protocols, originating from the results of the AVISPA project, and freely available on the project website.

The results of these experiments are given in the table available in Figure 3.3. We notice that some of the mutation operators are very likely to introduce security flaws in the considered protocols (exclusive-OR, homomorphism, hash functions). Other mutation operators are less efficient, even if they can still be applied in very specific cases (public keys, substitution, permutation). Even if the number of unsafe mutants remains relatively small (6% of the total), it is also interesting to notice that many protocols are not sensitive to the mutations we have considered.
### Figure 3.3: Results of Mutant Filtering

<table>
<thead>
<tr>
<th>Mutation / Result</th>
<th>Safe</th>
<th>Unsafe</th>
<th>Incoherent</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential</td>
<td>30</td>
<td>0</td>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>Exclusive-OR</td>
<td>17</td>
<td>13</td>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>Homomorphism</td>
<td>18</td>
<td>15</td>
<td>0</td>
<td>33</td>
</tr>
<tr>
<td>Public key</td>
<td>45</td>
<td>2</td>
<td>0</td>
<td>47</td>
</tr>
<tr>
<td>Substitution</td>
<td>286</td>
<td>5</td>
<td>134</td>
<td>425</td>
</tr>
<tr>
<td>Hash Functions</td>
<td>47</td>
<td>24</td>
<td>8</td>
<td>79</td>
</tr>
<tr>
<td>Permutation</td>
<td>420</td>
<td>1</td>
<td>4</td>
<td>425</td>
</tr>
<tr>
<td>Total</td>
<td>863</td>
<td>60</td>
<td>152</td>
<td>1075</td>
</tr>
</tbody>
</table>

**Contributions.** The first contribution of this work is a set of mutation operators dedicated to security protocols, written in HLPSL. The second contribution is a tool that implements these mutation operators and applies them on a verified HLPSL protocol file in order to produce a set of mutants.

### 3.1.2 Testing authorization systems

Authorization systems allow for the specification of access control policies which rule various protection aspects such as: the data confidentiality level, the procedures for managing data and resources, and the classification of data and resources into category sets with different access controls.

In this work, we focus on testing the implementation of the PolPA authorization system [91] and specifically on its Policy Decision Point (PDP). PolPA supports Usage Control (UCON) and history-based control, i.e. it checks the sequence of security relevant actions performed by the user in order to prevent or even interrupt the execution of an action when the policy is not satisfied. The PDP is the component that performs the decision process to decide whether an access should be allowed. Implementing the PDP is a critical activity for developers since any error in its implementation could seriously impact the decision process, authorizing accesses that should be forbidden by the policy and denying accesses that should instead be authorized.

To prevent these problems a rigorous and accurate verification and testing process must be adopted. Thus we first identify the main problems of the PDP implementation, i.e. we define a fault model; then from a given policy (gold policy), we derive a set of faulty policies according to the defined classes of problems and we generate the test cases able to detect the seeded faults, i.e. the selection of usage requests able to evidence a misinterpretation of authorization rules. Finally, we execute the test cases on a PDP implementation compliant with the gold policy and evaluate the obtained test results against the expected output.

We have realized a framework that automatically generates the test cases (i.e., authorization requests) starting from a predefined fault model. The key ideas of the testing process are: defining the fault model in terms of changes applicable to the PolPA policy, so that the modified policy versions can be used for testing purposes; parsing the PolPA policy in agreement to its rules, so to collect the information useful for the systematic derivation of the combinations of elements and values representing the different user's access modes, i.e test cases. A description of the main issues encountered during the framework realization is detailed in [27].

Thus, given a PolPA policy, the testing framework first generates the faulty policies according to a PolPA specific fault model; then, by applying the parsing to the policy and its mutated versions, it generates a test suite that covers the fault model so to exercise some specific aspects of the policy implementation and to highlight misbehaviors of the PDP implementation. Note that, for the moment, in this scenario we did not consider in the fault model problems related to additional interactions among authorization system components because we focused mainly on the possible faults of the PDP.

**PolPA authorization system.** The PolPA security policy exploits some composition operators (seq, or, par) to define the allowed behavior, i.e., the order in which security relevant actions can be performed. For a detailed description of the PolPA language we refer the reader to [91].
The architecture of the authorization system that enforces PolPA policies comprises Policy Enforcement Points (PEPs), a Policy Decision Point (PDP), a Policy Information Point (PIP), and a Policy Administration Point (PAP), as shown in Figure 3.4.

The PEPs should be integrated in the software components that implement security relevant actions to intercept their execution. The \texttt{tryaccess}(s,o,r) command (where \(s\) is the subject, \(o\) is the resource and \(r\) the action) is sent by the PEP to the PDP when the user tries to execute a security relevant action. The PEP allows the execution of the action only after a positive response from the PDP, represented by the \texttt{permitaccess}(s,o,r) command. Once an action has been permitted, the PEP should be able to detect when it terminates to issue the \texttt{endaccess}(s,o,r) command to the PDP.

The PDP performs the usage decision process. First, it gets the security policy from the repository managed by the PAP and builds its internal data structures for the policy representation. When it receives the \texttt{tryaccess}(s,o,r) command from a PEP, it checks the requested action against the security policy. Consequently, either the \texttt{permitaccess}(s,o,r) command is sent to the PEP, that executes the action, or the PDP returns \texttt{denyaccess}(s,o,r) to the PEP, that enforces it by skipping the execution of the access. Since it must keep track of the actions that are in progress, the PDP is also invoked by the PEP every time that an action that was in progress terminates, with the \texttt{tryaccess}(s,o,r) command. In fact, the PDP is always active because, if required by the policy, the PDP continuously evaluates a set of given authorizations, conditions and obligations while an action is in progress, and it could invoke the PEP to terminate this access through the \texttt{revokeaccess}(s,o,r) command. We focus on testing the PDP implementation, i.e. verifying that the PDP actually enforces its input security policy. The testing framework emulates a possible PEP by issuing the \texttt{tryaccess}(s,o,r) and \texttt{endaccess}(s,o,r) commands to the PDP, and by checking the returned answers. We assume that the input policy is correct, i.e., that does not contain errors or conflicting rules and expresses the security requirements of the resource owner who wrote it.

**Testing framework.** The testing framework consists of the following components (see [27] for further details):

**Fault Model Manager (FMM)** This component manages a predefined collection of possible types of faults that can occur during the evaluation of a PolPA policy due to incorrect control manipulation or violation of specific conditions.

**Policy Test Set Manager (PTSM)** This component collects the set of PolPA policies useful for testing purposes. 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 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) For each of the available policies (i.e. the policy and its faulty versions) this component automatically derives the test cases in terms of (sequence of) access requests.

Test Driver (TD) This component coordinates test case execution. In collaboration with the TCG it selects one by one the available test cases and, by simulating the PEP behavior and interacting with the PIP, transforms the test cases into tryaccess(s,o,r), and endaccess(s,o,r) commands.

Test Oracle (TO) This component compares the obtained PDP responses (that is, permitaccess(s,o,r), revokeaccess(s,o,r), or denyaccess(s,o,r)) with the given correct authorization replies associated to each of the generated test cases.

The FMM uses mutation operators for test case generation, rather than for test adequacy measurement and analysis. Thus, existing mutation operators are adapted for PolPA policies so that each faulty policy represents a syntactic fault that can be encountered during the PDP implementation. Next, we briefly describe the mutation operator classes:

Change Composition Operator (CCO) This class implements a violation of the order of execution of the commands sent by the PEP (tryaccess(s,o,r) and endaccess(s,o,r)). It is implemented by changing the composition operator. Specifically, let CO be the bag of composition operators (seq, or, par) included in the policy, the number of mutants is equal to the number of composition operators in CO times two.

Change Command (CC) This mutation operator class implements faults in the execution of a command sent by the PEP. It is implemented by changing the command. Specifically, let C be the bag of commands (tryaccess(s,o,r) and endaccess(s,o,r)) included in the policy, the number of mutants is equal to n × n-1 (n is the cardinality of C).

Change Guard String Predicate (CGSP) This class implements a wrong management of the values of string parameters. It is implemented by changing the predicate involving string parameters. Specifically, let S be the bag of PolPA predicates involving string parameters (sequal, startwith, scontains) included in the policy, the number of mutants is equal to the number of predicates times two.

Change Guard Integer Predicate (CGIP) This class implements a wrong management of the values of integer parameters. It is implemented by changing the predicate involving integer parameters. Specifically, let I be the bag of PolPA predicates involving integer parameters (iequal, morethan, lessthan) included in the policy, the number of mutants d is equal to the number of predicates times two.

Empirical evaluation. We applied the proposed framework to an exploratory policy simulating a generic situation in which the system under test (SUT) is the PDP implementing a policy specification that is considered correct. In this experiment the application of the fault model to the selected policy derived 78 mutated policies. We report the number of derived mutated policies for each mutant class in the second column of Table 3.1. TCG generated 2 test cases from the original policy and 45, 112, 8 and 8 test cases from the policies mutated according to the CCO, CC, CGSP, and CGIP mutant classes, respectively (third column), for a total of 175 test cases. Finally, each of the test cases has been executed on the PDP and

<table>
<thead>
<tr>
<th>Policy</th>
<th># Mutants</th>
<th># Test Cases</th>
<th># Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCO</td>
<td>14</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>CC</td>
<td>56</td>
<td>112</td>
<td>9</td>
</tr>
<tr>
<td>CGSP</td>
<td>4</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>CGIP</td>
<td>4</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>78</strong></td>
<td><strong>175</strong></td>
<td><strong>9</strong></td>
</tr>
</tbody>
</table>

Table 3.1: Experimental DATA
Figure 3.5: Number of use-after-free (left) and double-free (right) vulnerabilities in the CVE database in the 2008-2011 period, split by vulnerabilities in browsers, OSes, and other programs.

the responses have been collected and compared with the expected ones. The last column reports the comparison results: all responses were the expected ones, except those related to the test cases derived from the policies mutated according to the CC mutant class (thus a 0 in the column labeled # Faults). This means that the PDP implementation compliant with the gold policy did not contain any of the related faults. A different situation has been experienced for test cases derived from the policies mutated according to the CC mutant class. In this case, for 9 of the 112 test cases executed, the responses obtained were not the expected ones. The anomalies in the test case responses pointed out a problem in the PDP implementation due to the management of multiple actions. Although simple, this exploratory example represents a real application context and the fault discovered an important limitation of the considered PDP implementation.

For further details on this work we refer the reader to our publication [27].

Contributions. The implementation of an authorization system is a difficult and error-prone activity that requires a careful verification and testing process. In this paper, we focus on testing the implementation of the PoIPA authorization system and in particular its Policy Decision Point (PDP), used to define whether an access should be allowed or not. Thus exploiting the PoIPA policy specification, we present a fault model and a test strategy able to highlight the problems, vulnerabilities and faults that could occur during the PDP implementation, and a testing framework for the automatic generation of a test suite that covers the fault model. Preliminary results of the test framework application to a realistic case study are presented.

3.1.3 Early detection of dangling pointers

A dangling pointer is created when the object a pointer points to is deallocated, leaving the pointer pointing to dead memory, which may be later reallocated or overwritten. Dangling pointers critically impact program correctness and security because they open the door to use-after-free and double-free vulnerabilities, two important classes of vulnerabilities where a program operates on memory through a dangling pointer.

Use-after-free and double-free vulnerabilities are exploitable [7, 56] and are as dangerous as other, better known, classes of vulnerabilities such as buffer and integer overflows. Use-after-free vulnerabilities are particularly insidious: they have been used to launch a number of zero-day attacks, including the Aurora attack on Google's and Adobe's corporate network [126], and another 3 zero-day attacks on Internet Explorer within the last year [3, 47, 109].

Our analysis of the publicly disclosed use-after-free and double-free vulnerabilities in the CVE database [46] reveals two disturbing trends illustrated in Figure 3.5: (1) the popularity of use-after-free vulnerabilities is rapidly growing, with their number more than doubling every year since 2008 (over the same period
the total number of vulnerabilities reported each year has actually been decreasing), and (2) use-after-free and double-free vulnerabilities abound in web browsers (69%) and operating systems (21%), which use complex data structures and are written in languages requiring manual memory management (e.g., C/C++).

Use-after-free and double-free vulnerabilities are difficult to identify and time-consuming to diagnose because they involve two separate program events that may happen far apart in time: the creation of the dangling pointer and its use (dereference or double-free). In addition, understanding the root cause may require reasoning about multiple objects in memory. While some dangling pointers are created by forgetting to nullify the pointer used to free an object (non-sharing bugs), others involve multiple objects sharing an object that is deallocated (sharing bugs).

Sharing bugs happen because not all parent objects know about the child deallocation. They are particularly problematic for web browsers, which are built from components using different memory management methods. For example, in Firefox, JavaScript objects are garbage-collected, XPCOM objects are reference-counted, and the layout engine uses manual management. This mixture makes it extremely difficult to reason about objects shared between code using different memory management methods, which are particularly susceptible to dangling pointers bugs.

Previous work on tools for identifying and diagnosing use-after-free and double-free vulnerabilities [77, 105, 123] and on techniques to protect against their exploitation [8, 26, 54, 55, 102] focus on the use of the dangling pointer, which creates the vulnerability. In this work, we propose a novel dynamic analysis approach for analyzing and protecting against use-after-free and double-free vulnerabilities. Our approach shifts the focus from the creation of the vulnerability (i.e., the dangling pointer use) to the creation of the dangling pointers at the root of the vulnerability. We call our approach early detection because it identifies dangling pointers when they are created, before they are used by the program. Early detection is useful for different applications that target use-after-free and double-free vulnerabilities. In this work we evaluate early detection for testing for unsafe dangling pointers and for vulnerability analysis.

Testing for unsafe dangling pointers. A dangling pointer is unsafe if it is used in at least one program path and latent if the program never uses it. Use-after-free and double-free vulnerabilities are difficult to detect during testing because in a given execution the unsafe dangling pointer may not be created or it may be created but not used. Coverage can be increased using automated input generation techniques [36, 68, 96]. However, if the input space is large it can take a long time to find an input that creates the dangling pointer and triggers the vulnerability. Early detection extends the effectiveness of testing by also detecting unsafe dangling pointers in executions where they are created but not used. To identify at runtime unsafe dangling pointers and minimize false positives, we use the intuition that long-lived dangling pointers are typically unsafe. Moreover, long-lived dangling pointers are always dangerous and should be removed, even if currently not used, because modifications to the code by the (unaware) programmer may result in new code paths that use the (dangling) pointer. To identify long-lived dangling pointers, early detection tracks the created dangling pointers through time, flagging only those dangling pointers that are still alive after a predefined window of time.

Vulnerability analysis. A common debugging task is, given an input that causes a crash or exploitation, determining how to best patch the vulnerability, which requires understanding the vulnerability type and its root cause. Such crashing inputs are typically found using automatic testing techniques [36, 68, 96] and are usually included in vulnerability disclosures to program vendors. Our early detection technique automatically determines if a crash is caused by a use-after-free or double-free vulnerability and collects diagnosis information about the program state at both creation and use time. State-of-the-art memory debugging tools [77, 105, 123] provide information about the program state when the dangling pointer is used, but provide scant information about the dangling pointer creation (limited to the deallocation that caused it, if at all). This is problematic because a patch needs to completely eliminate the dangling pointer. If a patch only prevents the dangling pointer use that causes the crash, it may be incomplete since a dangling pointer may be used at different program points and there may be multiple dangling pointers created in the same bug. Furthermore, current tools offer little help when debugging sharing bugs, as these bugs require reasoning about multiple objects that point to the deallocated object at creation time. Our early detection technique tracks all pointers that point to a buffer, automatically identifying all dangling
pointers to the deallocated buffer, not only the one that produces the crash. Thus, it offers a complete picture about the type of dangling pointer bug and its root cause.

Undangle. We implement our early detection technique in a tool called Undangle that works on binary programs and does not require access to the program's source code. However, if program symbols are available, the results are augmented with the symbol information. We evaluate Undangle for vulnerability analysis and testing for unsafe dangling pointers. First, we use it to diagnose 8 vulnerabilities in real-world programs including four popular web browser families (IE7, IE8, Firefox, Safari) and the Apache web server. Undangle produces no false negatives and uncovers that two use-after-free vulnerabilities in Firefox were caused by the same dangling pointer bug. The reporter of the vulnerabilities and the programmers that fixed them missed this, leaving the patched versions vulnerable to variants of the attacks. We identify this issue with no prior knowledge of the Firefox code base, which shows the value of early detection for diagnosing the root cause of the vulnerability. Then, we test two recent Firefox versions for unsafe dangling pointers. Early detection identifies 6 unique dangling pointer bugs. One of them is potentially unsafe and we have submitted it to the Mozilla bug tracker. Our disclosure has been accepted as a bug and is pending confirmation on whether it is exploitable. Two other bugs are in a Windows library, so we cannot determine if they are unsafe or latent. The other three bugs are likely latent but our results show that the diagnosis information output by Undangle makes it so easy to understand and fix them, that they should be fixed anyway to close any potential security issues.

Contributions. This work makes the following contributions:

- We propose early detection, a novel approach for finding and diagnosing use-after-free and double-free vulnerabilities. Early detection shifts the focus from the creation of the vulnerability to the creation of the dangling pointers at the root of the vulnerability.

- We have designed an early detection technique that identifies dangling pointers when they are created. It uncovers unsafe dangling pointers in executions where they are created but not used, increasing the effectiveness of testing. When diagnosing a crash caused by a dangling pointer, it collects extensive diagnosis information about the dangling pointer creation and its use, enabling efficient vulnerability analysis.

- We have implemented our early detection technique into a tool called Undangle that works directly on binary programs. We have evaluated Undangle for vulnerability analysis using 8 real-world vulnerabilities and for testing on two recent versions of the Firefox web browser.

For further details on this work we refer the reader to our publication [35].

3.2 Runtime verification

Laws, inter-business contracts, security policies, and similar normative regulations define compliance requirements that IT systems need to enforce. For example, IT systems in US hospitals must enforce HIPAA, which regulates the dissemination of medical records and the subsequent obligations that medical staff are expected to fulfill. For banks, separation-of-duty constraints should reduce the risk of fraud. Data-usage contracts between different businesses regulate how sensitive documents are exchanged and subsequently disposed. Checking whether implemented IT systems comply with a body of regulations or policies is a problem of growing importance, since non-compliant behavior can lead to serious security breaches, monetary penalties, and the erosion of stakeholder's internal standards and commitments.

Monitoring techniques such as the ones developed by the runtime-verification community that observe and check system behavior during runtime offer a promising approach for compliance checking of IT systems, that is, checking whether the behavior of users and processes complies with a policy. This checking can either be done on-line while the system is running or off-line, such as during an audit by inspecting logged system actions. Moreover, most enforcement mechanisms, that is, mechanisms for preventing policy violations use some form of monitoring techniques that observe system behavior on-line.
3.2.1 Compliance checking with incomplete knowledge

In complex IT systems (usually composed of numerous interacting subsystems), the problem of incomplete knowledge about performed actions arises. In particular, logs, which record the policy-relevant system actions, may contain gaps due to corrupted files, logging-mechanism crashes, network failures, and so forth. Furthermore, when multiple logs are required to verify compliant behavior, they may disagree whether certain actions took place. For example, sharing a sensitive document between two parties may require the recipient to fulfill certain obligations. Thus, when analyzing the recipient's and the sender's logs against this policy, we need to treat all disagreements over the transfer of the document as incomplete knowledge, since favoring one log over the other may result in missed violations or false positives. Most runtime monitors, however, do not distinguish between a gap and a non-occurrence of an event. Thus applying them to incomplete logs can yield wrong results. For example, consider a policy like “a subject can access a document if the subject is not blacklisted”. If it is unknown whether a subject is blacklisted, then the subject is incorrectly reported as compliant.

In [19], we present a policy language and an accompanying monitoring algorithm that accounts for possibly incomplete and disagreeing logs. At the core of our approach is a three-valued truth space [83]. In addition to the classical Boolean values \( t \) (true) and \( f \) (false), which respectively represent the occurrence and non-occurrence of an event, we represent a knowledge gap about an event’s occurrence by the third truth value \( \perp \). Furthermore, when evaluating policies, their interpretation is as follows: the Boolean values \( t \) and \( f \) correspond to policy compliance and policy violation and \( \perp \) represents an inconclusive answer, which can be due to knowledge gaps of event occurrences or disagreeing events.

Our policy language is a variant of a first-order temporal logic [20, 40]. First-order temporal logics have been a good fit in various case studies for formally expressing and monitoring compliant policies [16, 73]. Special care must be taken when defining the semantics of a logic with additional truth values besides the classical Boolean values. In particular, a vital requirement for monitoring incomplete and disagreeing logs is to ensure that reported violations cannot be retracted if or when the log is eventually completed, for example, by recovering lost files. Otherwise, these results are of no value. More precisely, formalized policies must be monotonic with respect to the underlying partial ordering on knowledge, i.e., \( \perp \) is less than \( f \) and \( t \), and \( f \) and \( t \) are incomparable [25, 29, 61]. Our policy language guarantees this monotonicity requirement. Furthermore, the third truth value \( \perp \) is a first-class citizen at the object-level of our policy language: the classical logical connectives are extended to the three-valued truth space and there are specific connectives that guarantee expressive completeness with respect to the set of knowledge-monotonic operators. Such monotonic operators are needed in our application context to express at the logic’s object-level how disagreements between logged events should be resolved.

The monitoring algorithm presented in [19] for this three-valued setting is inspired by our previous monitoring algorithm [17, 20] for the standard Boolean setting. It iteratively scans the logged actions and soundly reports violations, i.e., whenever a violation is reported, it indeed is a policy violation. It also soundly reports potential violations, i.e., depending on how the knowledge gaps are filled, these might turn out to be real policy violations. However, our monitoring algorithm is not complete in the sense that some policy violations might not be reported. This limitation stems from the expressivity of our policy language over infinite domains. However, for an expressive fragment, which retains all the language’s connectives but limits the usage of free variables within a formula, we show that our monitoring algorithm guarantees completeness.

Contributions. We provide a solution to the problem of checking policy compliance in the presence of logging failures and disagreements between logged events. Our solution comprises a policy language and a monitoring algorithm. The policy language supports reasoning with incomplete knowledge. The monitoring algorithm may be used either off-line (for audit) or on-line (at runtime), and reports all policy violations and potential policy violations for an expressive fragment of our language. Although several features of our solution are present in related work, combining them to solve the stated problem is novel. In particular, our language is the first compliance language to consider three truth values at the object level, and our monitoring algorithm is the first algorithm to guarantee both soundness and completeness in a three-valued first-order setting.

For further details on the above work we refer the reader to our publication [19].
3.2.2 On the enforceability of security policies

Most conventional enforcement mechanisms are based on some form of execution monitoring. [110] began the investigation of which kinds of security policies can be enforced this way. In Schneider’s setting, an execution monitor runs in parallel with the target system and observes the system’s actions just before they are carried out. Whenever an action would result in a policy violation, the enforcement mechanism terminates the system. Note that there are alternative notions of enforceability where, rather than terminating execution just prior to a violation, one may raise exceptions, take corrective actions, and the like. For example, Ligatti and others [86, 87] consider enforcement mechanisms that can also insert actions into and delete actions from non-compliant behavior. Moreover, their enforcement mechanisms may even change compliant behavior, provided that the resulting behavior is semantically equivalent.

In [18], we refine Schneider’s setting, thereby overcoming several limitations. To explain the limitations, we first summarize Schneider’s findings. Schneider [110] shows that only those security policies that can be described by a safety property [10] on traces are enforceable by execution monitoring. Roughly speaking, (1) inspecting the sequence of system actions is sufficient to determine whether it is policy compliant and (2) nothing bad ever happens on a prefix of a satisfying trace. Note that a trace property must also be a decidable set to be enforceable. History-based access-control policies, for example, fall into this class of properties. Furthermore, Schneider defines so-called security automata that recognize the class of safety properties and that “can serve as the basis for an enforcement mechanism” [110, Page 40]. However, Schneider’s conditions for enforceability are necessary but not sufficient. In fact, there are safety properties that are not enforceable. This is already pointed out by Schneider [110, Page 41].

We provide a formalization of enforceability for mechanisms similar to Schneider’s [110], i.e., monitors that observe system actions and that terminate systems in case of a policy violation. A key aspect of our formalization is that we distinguish between actions that are only observable and those that are also controllable: An enforcement mechanism cannot terminate the system when observing an only-observable action. In contrast, it can prevent the execution of a controllable action by terminating the system. An example of an observable but not controllable action is a clock tick, since one cannot prevent the progression of time. With this classification of system actions, we can derive, among others, that availability policies with hard deadlines (requiring that requests are processed within a given time limit), are not enforceable although they are safety properties. Another example is administrative actions like assigning roles or permissions to users. Such actions change the system state and can be observed but not controlled by most (sub)systems and enforcement mechanisms. However, a subsystem might permit or deny other actions, which it controls, based on the system’s current state. Therefore the enforceability of a policy for the subsystem usually depends on this distinction.

In contrast to Schneider, we give also sufficient conditions for the existence of an enforcement mechanism in our setting with respect to a given trace property. This requires that we first generalize the standard notion of safety [10] to account for the distinction between observable and controllable actions. Our necessary and sufficient conditions provide a precise characterization of enforceability that we use for exploring the realizability of enforcement mechanisms for security policies. For different specification languages, we present decidability results for the decision problem that asks whether a given security policy is enforceable. In case of decidability, we also show how to synthesize an enforcement mechanism for the given policy. In particular, we prove that the decision problem is undecidable for context-free languages and PSPACE-complete for regular languages. Moreover, we extend our decidability result by giving a solution to the realizability problem where policies are specified in a temporal logic with metric constraints.

Contributions. We overcome limitations of Schneider’s setting on policy enforcement based on execution monitoring. First, we distinguish between controllable and observable system actions when monitoring executions. Second, we give conditions for policy enforcement based on execution monitoring that are necessary and also sufficient. These two refinements of Schneider’s work allow us to reason about the enforceability of policies that, for instance, involve timing constraints. We also provide results on the decidability of the decision problem of whether a policy is enforceable with respect to different specification languages.

For further details on this work we refer the reader to our publication [18].
3.2.3 Runtime enforcement of XACML policies

Over the last decade researchers have shown the advantages of attribute-based access control. Long-standing accesses (e.g., execution of an application in Grid, running of a virtual machine in Cloud) imply a continuous runtime monitoring of attribute values and the reassessment of access decision. The access should be verified continuously and allowed only if the attribute values comply with security policies.

This section describes an enhanced authorization system which is able to interrupt accesses that are in progress when the corresponding access rights do not hold any more [85]. Our approach is based on the Usage Control (UCON) model, defined by Sandhu and Park in [127], and on the OASIS XACML standard for access control.

UCON introduces new features in the decision process w.r.t. traditional access control models, such as mutable attributes of subjects, objects, and environment, and the runtime policy enforcement to guarantee that the right of a subject to use the resource holds while the access is in progress. The decision process consists of two phases. The pre-decision phase corresponds to traditional access control, where the decision process is performed at the request time to produce the access decision. The ongoing decision phase, instead, is executed after the access is started and implements the continuity of control that is a specific feature of the UCON model. Continuous control implies that policies are re-evaluated each time mutable attributes change their values. If the decision process detects a policy violation, the access is revoked and resources are released. Also, the requester can end the access at his/her discretion.

In recent years UCON has drawn a significant interest from the research community on policy formalization and enforcement. There have been several attempts to implement usage control, while the realization based on existing security standards is still an open issue. An efficient and flexible framework (i.e., a policy schema, an architecture and an implementation) for access and usage control based on the OASIS XACML standard is a challenge we address in our work.

In [44], we present the U-XACML policy language obtained by extending the XACML language with constructs for usage control. To represent continuous control, the U-XACML specifies when the access decision must be taken through the clause DecisionTime in the <Condition> elements (the admitted values are pre and on denoting, respectively, pre and ongoing decisions), and the TriggerOn clause in the <Obligation> elements. To represent attribute updates, it defines a new element, <AttrUpdates>, that contains a collection of single <AttrUpdate> elements to specify update actions. The time of update is stated by the element UpdateTime that has values of pre, on, and post.

Figure 3.6: U-XACML Policy Enforcement Architecture

Figure 3.6 shows the architecture of the authorization system supporting the runtime enforcement of U-XACML policies. As in most authorization systems, the main components are:

**Policy Enforcement Point (PEP)** is integrated within the security component that executes the security relevant actions that are regulated by the security policy. The PEP must be non-bypassable and tamper-proof. It must also intercept the invocation and suspend the execution of the security relevant actions, communicate with the UCON authorization system, enforce access decisions by resuming or preventing the execution of the suspended actions, intercept the normal termination of the security relevant actions, and interrupt the execution of the security relevant actions if requested by the UCON authorization system.
Policy Information Point (PIP) provides an interface for retrieving mutable attributes needed by the UCON authorization system to produce access decisions. The PIP contacts Attribute Managers (AMs) to obtain the required attributes. The AMs are attribute repositories which provide facilities for storing attributes, keeping updated, etc.

Context Handler (CH) is the front-end of the UCON authorization system, that manages the protocol for communicating with PEPs and PIPs. It converts and forwards messages sent between components in the proper format. During the pre-decision phase, the CH also contacts the PIP and retrieves security attributes.

Policy Administration Point (PAP) stores and manages U-XACML policies.

Policy Decision Point (U-PDP) evaluates U-XACML policies to produce access decisions for access requests.

Access Table (AT) keeps meta-data regarding accesses in progress, i.e., usage sessions. It contains a table holding the current sessions with their status (i.e., pending, active, ended, revoked), a table with the identifiers of the attributes needed to service each session, and a table with the latest values for these attributes.

Session Manager (SM) is the main novelty of the architecture. It manages usage sessions. The SM creates a new entry in the AT for each new tryaccess that is allowed by the U-PDP, i.e., for each new usage session. The SM also manages the ongoing decision phase. It evaluates the access right before the access starts, and performs the periodical retrieval of mutable attributes. When the values of some attributes change, it triggers the access right re-evaluation of each usage session that exploits these attributes. If two usage sessions refer to the same attributes, the SM queries those attributes just once instead of sending two disjoint queries. Then, the SM triggers the access re-evaluation for both sessions with the new values. In fact, attribute queries are unique for all sessions. This decreases the security overhead due to concurrent usage sessions. If the AM supports a subscription, the SM performs the subscription for relevant attributes through the PIP. Instead, for AMs that do not support a subscription, the SM repeatedly queries the AM through the PIP for detecting changes in attribute values. The time interval between two consecutive queries is a configuration parameter and is set according to the attribute to be monitored. There are also some attributes which are updated as the result of the policy enforcement. These attributes do not require to be retrieved and the SM stores and updates them directly in the AT.

We present the message workflow in the case that a session is revoked due to the policy violation. The normal end of access has a similar workflow and thus it is omitted. Figure 3.7 shows the message workflow between main components of the architecture. The first message, tryaccess, is sent by the PEP to the CH when the request for the execution of a security relevant action is intercepted by the PEP. The CH retrieves the values of the attributes that could be relevant to the decision process by sending the attr query message to the PIP that, in turn, contacts the relevant AMs exploiting their specific protocols, and sends back these values to the CH through the message attr value. The CH then sends the access request that includes the attributes previously collected, to the U-PDP. The U-PDP loads the U-XACML policy from the PAP. The U-PDP evaluates the policy and replies with the response to the CH.

Let us suppose that the policy allows the execution of the requested action: then the CH replies with the permitaccess message to the PEP. Before sending it, the CH sends the create entry to the SM for creating a new entry that represents the new usage session in the AT. Also, the create entry message contains attribute updates which should be performed by the SM before the usage session starts.

When the access has begun, the PEP sends the startaccess message to the CH, that sends the message update entry to the SM. The SM contacts the AT to change the status field of the database entry related to this usage session from pending to active, and triggers the evaluation of the ongoing access for the first time. Hence, the SM starts the continuous policy re-evaluation loop and sends the attr query message to the PIP to get the fresh values of attributes that are relevant for this access. The PIP gets these values from the AMs, and sends them back to the SM. If one of the collected values is different from the one cached in the AT, the SM contacts the CH sending the policy reevaluation message; the
Figure 3.7: Sequence Diagram of Usage Control Enforcement

CH translates it in the right format and sends the request message to the U-PDP that performs the re-evaluation of the access right. If the decision included in the response message is permit, then the CH forwards this answer to the SM. The SM performs ongoing attribute updates contained in the U-PDP’s reply and continues the policy enforcement by collecting fresh attributes and triggering the access re-evaluation. Instead, if the response included in the response message sent by the U-PDP is deny, the SM sends the revokeaccess message including the data to identify the right PEP to the CH that forwards it to this PEP that will force the access revocation, e.g., by releasing the resource.

Contributions. We have presented an enhanced authorization framework that is able to deal with long lasting accesses, preventing users from continuing the usage of previously assigned resources when their access rights are no longer valid. We have revised the XACML policy schema for expressing usage control scenarios. We have enhanced the XACML reference architecture for supporting a run-time enforcement of usage control and introduced state-full communications between the PEP and the PDP. Finally, we have proposed a proof-of-concept implementation of our framework based on web-services standards, which has shown promising performance results.

For further details on this work we refer the reader to our publication [85].
4 Transverse Methodology

Security concerns are specified at the business level but have to be implemented in complex distributed and adaptable systems of Future Internet services. We need comprehensive assurance techniques in order to guarantee that security concerns are correctly taken into account through the whole SDLC. First, model-based testing enhances security testing techniques by combining the typical code level testing with models pertaining to the early stages of the SDLC. These techniques are covered in Section 3.1. Second, we have worked towards bridging the gap between model-driven security (at the design level) and language-based security (at the implementation level). The interaction between these two fields was hitherto rather limited.

4.1 Bridging model-based and language-based security

4.1.1 Non-interference on UML state-charts

Non-interference is a concept that allows one to reason about the security of systems with respect to information flow policies for different groups of users. The rough idea is that what users do in one security level does not affect what other users see or may do at other levels. But this is a semantically defined property, it is a priori not clear what syntactical properties of specifications correspond to it. Many of the security problems of implementations could be already spotted at design time if information flow would be a concern in early phases of software development and there would be methods and tools to deal with non-interference in these phases, without expecting a high-level of mathematical sophistication from a regular software developer or security expert at the design-level.

In this work, we propose a light-weight, automatic strategy for verifying non-interference in the composition of services whose behavior is described by deterministic UML state-charts. The method is based on so-called unwinding theorems, extended to cope with the particularities of state-charts: the use of variables for keeping history of the state, guards for transitions, actions, and hierarchical states. Moreover, we aim at verifying systems where the interaction between different components plays a fundamental role. To achieve this, we discuss sufficient conditions for deciding on the security of the composition of already verified components. This is a key factor in the scalability of the approach, very important for the success of verification in realistic settings.

This kind of analysis (first introduced by Goguen and Meseguer in 1982 [69]) is mathematically defined over the inputs and outputs visible to groups of users. Its main advantage over other security analysis is that it allows to pin down subtle flows of information, usually known as covert channels, that are difficult to spot when focusing merely on analyzing security mechanisms (such as access control mechanisms). Information flow properties assume perfect access control to guarantee that different groups of users do not see certain inputs and outputs of other users directly. This is a reasonable assumption since attackers usually exploit the information that is shared by victims through common interfaces instead of trying to break directly the access control mechanisms.

In the past decades, different information flow properties have been proposed for coping with different system models such as non-deterministic systems, distributed systems, and imperative programming languages. At the abstract level, results about compositionality and refinement have been published for many security properties [89, 90, 104]. In the "language-based" realm (i.e. analysis of source code) mature tools for information flow analysis on annotated code exist (e.g., [1, 2, 67]). All of these are indeed promising steps towards the industrial application of the fine-granular analysis offered by the property-centric point of view of information flow.

To validate our theoretical results, we report on a prototype implementation that automatically verifies models where our unwinding theorem is applicable. In Deliverable D11.3 [118], we apply this implementation to examples motivated by a case study from the Smart Grid domain. The case study allows us to discuss and validate the utility of our approach.

Non-interference. Assume a system is a deterministic black-box transforming sequences of input events $I$ into sequences of output events $O$ by means of a semantics function: $[,]: I \rightarrow O$. We further assume, for simplicity, there are only two types of users: high users $H$ and low users $L$. The sets of input and
output events are also marked as low or high, depending on what a low user is allowed to see. A sequence of input and output events can then be filtered according to what a low user is allowed to see: $\tilde{\cdot}L$. Non-interference is the property that for all input sequences $\tilde{\cdot}$ it is true that $[\tilde{\cdot}L]_L = [\tilde{\cdot}L]_L$. In other words, the output seen by the lower users is independent of the input by higher users, up to the point that is not even noticeable whether the high users perform any action on the system. A stronger version of this property is usually used in the language-based information flow analysis domain [15,66,74].

State-charts. A Mealy machine [95] is depicted as a directed graph with labeled transitions of the form $\alpha/\beta$, representing that the input event $\alpha$ triggers the output event $\beta$. A Mealy machine induces a semantics $\cdot$ in a natural way. If we further divide the input and output events into high an low events, we can apply the definition of non-interference to a Mealy machine.

To deal with the so-called ‘state-explosion’ problem, which arises when the number of states and transitions increases, formalisms have been proposed that include the notion of sub and super-states. Quite prominently, Harel [76] proposed the notion of state-chart that has been used as the basis for UML. This is basically an extension to Mealy machines that includes:

- **Hierarchical states**: Single states can contain sub-states and transitions among the sub-states up to arbitrary depth.
- **Clustering**: A graphical convention to group events that trigger a transition to the same state in a group of states.
- **Orthogonality and concurrency**: Multiple sub-state-charts can be modeled as concurrent processes.

**Non-interference verification for UML state-charts à la UMLsec.** UML has adopted an extension of Harel’s state-charts to represent the behavior of classes. It allows a list of actions as a consequence of an event, including calling methods, updating variables, and outputting values. In this work we will restrict to a fragment of UML state-charts defined as follows:

- **Input events** labeling transitions can be either methods of the associated class with concrete parameters or with variables to represent calls with different parameters or global system events (like the ticks of a system clock).
- **Actions** associated to an input event can be either outputting an event (written return event) or a variable assignment, where the variables are attributes of the associated class or parameters of the input.
- **Guards** are decidable conditions on the input parameters or the values of the attributes.

We restrict to a subset of UML state-charts, following the notation of [81], but considering only deterministic automata, resulting in a simpler semantics.

One possibility to verify UML state-charts would be to unfold and construct a semantically equivalent Mealy machine and use the classical unwinding theorems. However, this would be computationally quite expensive in general: the purpose of the UML state-chart notation is to avoid state and transition explosion. We have extended the unwinding theorem accordingly for coping soundly with these differences in the notation in an efficient way, by simultaneously extending an unwinding relation and constructing a set of tainted variables associated to each state, which keeps track of which variables are directly or indirectly dependent on high inputs, in the spirit of language-based information flow analysis.

Our main result may be stated as:

**Theorem 4.1** Let $U$ be a UML state-chart. If $U$ admits a relation $R$, together with a function tainted that associates a set of variables to each state $s \in U$, such that $(R, \text{tainted})$ is an unwinding, then $U$ respects non-interference.

The technicalities of the definition of unwinding (including tainting) and the proof of the theorem are omitted here, for details see [103].
Service composition. To reason about the security of a system that is built upon interacting objects, we would need to obtain composed state-charts out of the state-charts defining the single object’s behavior. The notion of compositionality we use is based on message passing between state-charts: the output messages generated by a state-chart \( A \) can be input messages for a state-chart \( B \) but not vice-versa. This corresponds formally to a special case of parallel composition [31,97], where we restrict the feedback only to occur in one direction. In other words, we do not allow callbacks, which are related to mutual recursion between objects.

For a given partition of the set of input \( I = I_H \cup I_L \) and output \( O = O_H \cup O_L \) alphabets of the composition \( A \otimes B \) there exist sufficient conditions on \( A \) and \( B \) such that this composition respects non-interference. This is notably not the case in general for information flow properties [89]. However, in our case we can derive a positive result:

**Theorem 4.2** Let \( I = I_H \cup I_L \) and \( O = O_H \cup O_L \) be a partition of the input and output alphabets of \( A \otimes B \). If non-interference holds for an extension of the policy in \( I \) and \( O \) to the unspecified events in \( I_B \cap O_A \) in \( A \) and \( B \), then non-interference holds on \( A \otimes B \).

Tool support and validation. In Deliverable D11.3 [118], we report on experiments made to implement the enhanced unwinding technique and the compositionality theorem and apply them on examples from our case study. We have implemented a prototype of the above algorithm in Haskell.

Related work. Starting with the work of Goguen and Meseguer [69], many information-flow properties have appeared for specific system models and to capture different notions of security. Rushby discusses unwinding theorems in a more modern notation [107] along with transitive vs. intransitive information flow policies. General unwinding theorems for a wide range of information flow properties have also been suggested by Mantel in [70]. Mantel has also unified most of these properties into a common framework, the Modular Assembly Kit for Security (MAKS) [90], also deriving new unwinding theorems. This work is also probably the best reference for a discussion on the different properties proposed for abstract non-deterministic and distributed system designs.

In the language-based world, different static approaches have been suggested for verifying information-flow properties, prominently type-based systems like Volpano-Smith [124] or more recently Barthe et al. [15]. Also works based on abstract interpretation and analysis of Program Dependency Graphs [66,74] give approximations to non-interference for JavaCard-bytecode and Java respectively. Tools for information flow analysis on annotated code using these techniques are for example Jif [1], JOANA [67] and STAN [2]. Jürjens [81] defined a stereotype for non-interference on state-charts that is equivalent to the notion used in this chapter for the deterministic case, but no verification strategy or compositionality results are discussed. In [9] Alghathbar et al. model flows of information with UML Sequence diagrams and Horn clauses. However, their focus is on high-level information flow policies where only actors and the messages their exchange are modeled, and no explicit relation between the information control rules and a semantic property is given. To the best of our knowledge, there exist no works extending unwinding theorems for UML state-charts that consider parametrized events, guards and actions with side-effects.

Contributions We have presented an efficient verification strategy for state-charts that is sound with respect to classical non-interference. Our technique is fully automatic and can help to narrow the gap between theory and practice for information-flow secure software development in an industrial context. Our results extend previous work in the area to a non-trivial subset of UML statecharts. On a technical level, we have shown how to link unwinding theorems defined in input/output state-machines with verification techniques related to the imperative programming language domain.

For further details we refer the reader to our publication [103].
5 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. The first research line is an investigation of different vulnerability predictors, applied in a case study to Android applications. The second research line focuses on quantitative access control, where we generalize access control decisions from boolean or three-valued domains to richer metric domains.

5.1 Predicting vulnerabilities in Android applications

The focus of this work is creating accurate vulnerability prediction models in the domain of Android applications. Our main goal is to use machine learning techniques in order to predict vulnerabilities by training the machine on historic data (e.g., vulnerabilities that were extracted from the previous versions of a given application). Repositories containing a large version history of open source mobile applications for the Android platform are readily available and represent an ideal testbed for our work. We have devised two different, but complementary tracks of investigation. The first approach leverages code metrics (such as, lines of code, number of attributes, etc), while the second approach relies on text mining for predicting vulnerabilities.

5.1.1 Application selection

The starting point for our work is the selection of Android applications, whereby we have selected 10 different applications. The rationale for selecting these applications was to have a diverse selection of Android apps in terms of size, number of revisions, category and popularity in terms of number of downloads. Table 5.1 shows the categorization of each application in terms of these characteristics. Throughout our empirical analysis, we have used several versions of each application spread over a period of approximately 1.5 years. The second column indicates the number of features in the feature vector that entails all files of the given version of the application. The LOC (lines of code) metrics represents the total number of lines of code including comments. The number of Java files presents the count of all files with the .java extension excluding libraries. Some applications make use of libraries in their raw source format rather than a packaged .jar file. Note that all these three metrics apply to the first version of each application that is used to build the prediction model. The last two columns of the table reflect on the type of the application and its popularity in terms of number of downloads.

All applications have increased in terms of size throughout the various versions we have used. Connectbot has grown only marginally (less than 10% growth); keepassdroid, mustard and FBReaderJ have grown moderately (growth rate between 20% and 50%); while the remaining applications have increased substantially (growth rate between 50% and 200%).

In order to assign the vulnerability labels we leverage the state-of-the-practice Fortify tool\(^1\) that analyzes the source code for various known types of software security vulnerabilities. Fortify not only spots a vulnerability, but also assigns a severity for each vulnerability found. In both approaches, we have treated a component as vulnerable if Fortify has assigned any type of vulnerability to it and as clean otherwise. Note that the definition of a component differs in our approaches as in the first approach a component represents a Java class, while in the second approach it represents a Java file.

5.1.2 Vulnerability prediction using code metrics

Our first approach relies on the use of object-oriented source code metrics as predictors of vulnerabilities [108]. We have used the JHawk 5 tool\(^2\) to collect about 40 different metrics for each class, including lines of code, coupling, cyclomatic complexity, depth of the inheritance tree, and so on. For each class the set of metrics is in fact the feature vector that is used by the machine learning algorithm to both train the

\(^1\)https://www.fortify.com
\(^2\)http://www.virtualmachinery.com/jhawkprod.htm
Table 5.1: Applications

<table>
<thead>
<tr>
<th>Application name</th>
<th># selected versions</th>
<th># Java files</th>
<th># of features</th>
<th>LOC</th>
<th>Category</th>
<th># downloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>mileagetracker</td>
<td>6</td>
<td>10</td>
<td>1092</td>
<td>10048</td>
<td>finance</td>
<td>100k - 500k</td>
</tr>
<tr>
<td>CoolReader</td>
<td>13</td>
<td>24</td>
<td>2342</td>
<td>5643</td>
<td>books &amp; reference</td>
<td>1000k - 5000k</td>
</tr>
<tr>
<td>Anki-Android</td>
<td>8</td>
<td>27</td>
<td>3008</td>
<td>6640</td>
<td>education</td>
<td>100k - 500k</td>
</tr>
<tr>
<td>boardgamegeek</td>
<td>8</td>
<td>28</td>
<td>1794</td>
<td>5007</td>
<td>book and reference</td>
<td>10k - 50k</td>
</tr>
<tr>
<td>Crosswords</td>
<td>17</td>
<td>36</td>
<td>2507</td>
<td>6333</td>
<td>brain &amp; puzzle</td>
<td>5k - 10k</td>
</tr>
<tr>
<td>connectbot</td>
<td>12</td>
<td>45</td>
<td>4625</td>
<td>11144</td>
<td>communication</td>
<td>1000k - 5000k</td>
</tr>
<tr>
<td>mustard</td>
<td>12</td>
<td>76</td>
<td>3162</td>
<td>10048</td>
<td>social</td>
<td>10k - 50k</td>
</tr>
<tr>
<td>K9 Mail</td>
<td>19</td>
<td>97</td>
<td>7767</td>
<td>32803</td>
<td>communication</td>
<td>1000k - 5000k</td>
</tr>
<tr>
<td>KeePass</td>
<td>13</td>
<td>114</td>
<td>2835</td>
<td>10366</td>
<td>tools</td>
<td>100k - 500k</td>
</tr>
<tr>
<td>FBReader</td>
<td>14</td>
<td>226</td>
<td>5274</td>
<td>24411</td>
<td>books &amp; reference</td>
<td>1000k - 5000k</td>
</tr>
</tbody>
</table>

(prediction model as well as use the feature vector to predict the vulnerability of a given class. We used Support Vector Machines (SVM) in order to build a classification model that predicts whether a Java class is vulnerable.

We have used only five versions of the FBReader in the context of our first approach as it is currently in a rather exploratory state. As a training set required to build the prediction model, we used the data (vulnerabilities and code metrics) from the first of our five versions of FBReader. The model built during the training phase was tested on the subsequent four versions. We assessed model performance (in terms of prediction power) by means of three indicators:

- **Accuracy** is the rate of correct results, i.e., the sum of true positives ($TP$, classes containing weaknesses that are correctly classified as vulnerable) and true negatives ($TN$, non-vulnerable classes that are correctly classified as negatives) over the total number of classifications.

- **Precision** is the probability that a class classified as vulnerable is indeed vulnerable. It is computed as the ratio $TP/(TP + FP)$, where $FP$ is the number of false positives, i.e., Java classes erroneously classified as vulnerable.

- **Recall** is the probability that a vulnerable class is actually classified as such. It is computed as the ratio $TP/(TP + FN)$, where $FN$ is the number of false negatives i.e., Java classes erroneously classified as not containing vulnerabilities.

Figure 5.1 shows the precision, recall, and accuracy of the prediction model over the four versions used for testing. The results of our first investigation show that is possible to build a (binary) prediction model with high accuracy and high precision, but low recall.

5.1.3 Vulnerability prediction using text analysis techniques

Our second approach takes a different perspective to the problem of vulnerability prediction [80]. Instead of relying on certain “cooked” features of the source code (such as, complexity and size metrics) as in the case of the first approach we have tried to simply use the source code as a vulnerability predictor itself.

The starting point for our approach is the source code of a software system that consists of a number of Java files. Each Java file is transformed into a feature vector where every word (also called a monogram in text processing) within that Java file is treated as a feature.

In order to transform the preprocessed source code into a feature vector, we need to tokenize the textual representation of the source into a set of monograms. As a set of delimiters we have chosen to
use not only white spaces, but also the Java “punctuation” characters (such as, “, ; ( ) { } [ ]”) as well as mathematical and logical operators (such as, “+ - / * | ∥ & && !”). In a feature vector each monogram (i.e., feature) must also have an assigned value. We use the count of a given monogram in a given Java file source code as its value.

Consider the figure 5.2 that depicts the HelloWorld.java file.

```java
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */

class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

Figure 5.2: Hello World Java file

The feature vector of the HelloWorld.java file is:

\[ \text{The: 1, HelloWorldApp: 2, class: 2, implements: 1, an: 1, application: 1, that: 1, simply: 1, prints: 1, Hello: 2, World: 2, to: 1, standard: 1, output: 1, public: 1, static: 1, void: 1, main: 1, String: 1, args: 1, System: 1, out: 1, println: 1, Display: 1, the: 1, string: 1} \]

where each of the monograms is followed by a count (in this case 1). Note that in this example we do not follow any particular (e.g., SVM) notation.

Our second approach was empirically evaluated on all 10 applications presented in the previous section. Similar to our first approach we have built a prediction model based on the first available version of each application. We have then used each prediction model to predict the vulnerabilities of the remaining versions of each application. Figures 5.3 and 5.4 present a combined graph illustrating the overall precision and recall for all versions of each application using SVM. We have replicated the same experiment.
also using classification and regression trees (CART) approach. The SVM technique achieves better precision, but lower recall compared to CART. CART seems to provide more uniform numbers. All in all both techniques achieve good performance in terms of precision and recall for most applications.

**Contributions** We have devised two different approaches for building vulnerability prediction models tailored for the Android applications. Our first approach starts from the assumption that certain source code metrics (e.g., lines of code, number of attributes, depth of inheritance tree) are likely to be predictors of whether a given component is vulnerable or clean. An empirical investigation has pointed out that the vulnerability prediction model built from source code metrics results in very high precision, but low recall. Our second approach relies on text mining techniques and treats each monogram in the source code as a potential vulnerability predictor. We have explored the potential of the bag-of-words representation and discovered that a dependable prediction model can be built by means of machine learning techniques. In a validation with 10 Android applications we have obtained performance results that often outclass state-of-the-art approaches.

For further details on these works we refer the reader to our publications [80, 108].
5.2 Quantitative access control

An important aspect of security metrics is to provide means to quantify the impact of security decisions, such that a security mechanism can make the best decision possible, rather than simply make a good one. This is particularly important for critical systems, where in some situations, the security mechanism might need to choose the lesser of two evils. For instance, consider an healthcare environment, when a patient is in a critical medical situation, and the attending physician is not available: in that case, the system can either allow a nurse to access the medical record of the patient, which is likely to be a violation of the basic security policy, or can deny the access, thus preventing the patient to receive the best possible treatment. Although most systems have overriding and delegation mechanisms to deal with such situations, this problem can be generalized into defining security mechanisms that consider the impact on the security and utility metrics of each decision rather than simply verifying whether a particular access belongs to a predefined list of authorized accesses.

The main problem we have been trying to address is therefore to specify an access control mechanism where the policy is no longer defined as a list of rules describing which accesses are authorized and/or denied, but instead where the states of the system are associated with a notion of utility, describing at the same time the usability and the security of the current state. The objective of the security mechanism therefore becomes to optimize the utility of the system, by selecting at each step the decision that will eventually lead the system to the best possible reachable state.

Our main contribution is the definition in [92] of an access control mechanism as a Markov Decision Process (MDP) [24], which are well-known mechanisms in decision theory tailored to tackle the kind of problem we want to address (i.e., optimizing the utility of a system). We present in the following this modeling, together with its implementation within the GNU Linear Programming Kit (GLPK)\(^3\), an open-source linear solver.

5.2.1 Access Control Markov Decision Process

We have expressed in [92], within a single framework, the notion of utility of an access, some decisions beyond the traditional allowing/denying of an access, the uncertainty over the effect of executing a given decision, the uncertainty over the current state of the system, and the optimization of this process for a (probabilistic) sequence of requests. We consider the following situation: the environment (consisting of users, processes, etc) submits access requests to the access control decision process, which makes a decision for each request.

Intuitively speaking, we consider the system as a state machine, where each state \(\sigma \in \Sigma\) consists of two separate parts: the security information (belonging to a general type \(I\)) and an access request to evaluate (belonging to a set of requests \(R\)). For instance, in an healthcare environment, the security information could represent the accesses currently done, the availability of the staff, the qualification of the staff, etc. Furthermore, we introduce a special request \(\epsilon\), which denotes that there is no request to evaluate. Hence, a transition is done from one state to another with a security decision (as decided by the security mechanism), corresponding to the evaluation of the request in the source state. We usually consider only two possible decisions, allow and deny, but more complex decisions can be included, such as those corresponding to the mitigation of an access or the launch of a security audit. The only requirement to add a new decision is to describe the effect on each state of this decision. Note that a possibly counter-intuitive aspect of this approach is the fact that requests are defined in the state in a non interactive way, that is, the decision process jumps from a state where a request is to be evaluated to another state where a new request is already specified. However, we usually address this aspect with the fact that transitions are probabilistic, which allows us to specify for each request the probability of the system to move to a state where this request is to be evaluated.

**Definition. 1** An Access Control Markov Decision Process (ACMDP) is a tuple \((\Sigma, A, P, W)\), where \(\Sigma = I \times R\) is a set of access control states, \(A\) is a set of decisions, \(P : \Sigma \times A \times \Sigma \rightarrow [0,1]\) is the probability function, such that \(P(\sigma_i, a, \sigma_j)\) stands for the probability of reaching the state \(\sigma_j\) by executing the decision \(a\) from the state \(\sigma_i\), and \(W : \Sigma \times A \times \Sigma \rightarrow U\) is the reward function (where \(U\) is the utility domain, defined

\(^3\)http://www.gnu.org/software/glpk/
for instance as $\mathbb{R}$ or $\mathbb{N}$, such that $\mathcal{W}(\sigma_i, a, \sigma_j)$ stands for the reward associated with executing the decision $a$ from the state $\sigma_i$ and arriving in the state $\sigma_j$. When no confusion can arise, we write $p_{ij}$ and $w_{ij}$ for $\mathcal{P}(\sigma_i, a, \sigma_j)$ and $\mathcal{W}(\sigma_i, a, \sigma_j)$, respectively.

A policy is a function $\delta : \Sigma \rightarrow \mathcal{A}$, which returns for each state the decision to execute. In other words, given a state $(\iota, r)$, $\delta(\iota, r)$ represents the decision about the request $r$ for the security information $\iota$. The goal is now to define the optimal policy, that is, the policy that optimizes the reward obtained by the system. For the sake of clarity, we can consider that a reward is a real number, and that optimizing the reward is achieved by maximizing it, i.e., the higher the reward is, the better.

A naive way to define an optimal policy could be to simply consider for each decision the sum of each corresponding reward weighted by its probability. In other words, given a state $\sigma_i$ and a decision $a$, we can define the notion of immediate reward $q_i^a = \sum_j p_{ij} w_{ij}$, where the sum is done over all states $\sigma_j \in \Sigma$. Hence, given a state $\sigma_i$ and two decisions $a$ and $b$, we would say that $a$ is better than $b$ if, and only if, $q_i^a \geq q_i^b$. However, such an approach only accounts for the short term impact of a security decision, and does not consider the impact of the future decisions that are to be made. For instance, it can be the case that $q_i^a \geq q_i^b$, but at the same time the decision $a$ only leads to states from which very small rewards can be obtained, while $b$ can lead to states from which high rewards can be obtained. This is a classical problem in Decision Theory, where it might be better to take a small immediate reward in order to eventually obtain a higher global reward (which is the rationale behind many financial investments).

Hence, we define the value of a state, which combines the immediate rewards obtained from this state and the values of the next states. Given a policy $\delta$ and a state $\sigma_i$, the value function $V^{\delta}$ is calculated with the Bellman equations [24]:

$$V^{\delta}(\sigma_i) = q_i^{\delta(\sigma_i)} + \beta \sum_{\sigma_j} p_{ij}^{\delta(\sigma_i)} V^{\delta}(\sigma_j)$$

(5.1)

where $0 \leq \beta \leq 1$ is a discount factor, enabling to put more or less weight on future decisions. Note that these are clearly recursive definitions, and that in some cases, they might not admit any solution. The typical technique to solve them is by linear programming, and we present an example in Sec. 5.2.3.

Assuming that the previous equations can be solved, we can derive the definition of an optimal policy $\delta^*$:

$$\delta^*(\sigma_i) = \arg\max_{a \in \mathcal{A}} [q_i^a + \beta \sum_{\sigma_j} p_{ij}^{a} V^{\delta^*}(\sigma_j)]$$

(5.2)

Intuitively, an optimal policy is a policy that for any state $(\iota, r)$ returns the decision for $r$ that maximizes in the long term the utility of the system.

### 5.2.2 Access Control Partially Observable Markov Decision Process

Another degree of uncertainty that should be considered is the uncertainty over the current state of the process. Indeed, although an ACMMDP can handle uncertainty in the future, it also assumes that the current state is known. However, this is not always the case in practice, especially in open and distributed systems, where information is collected from different, potentially unreliable sources. For instance, the meta-information about a file can be corrupted during a transfer, and the exact sensitivity of this file might not be known. Similarly, the exact location of a user might be imprecise, and only an estimation of her probable locations might be provided.

We introduce here the notion of an Access Control Partially Observable Markov Decision Process (ACPOMDP), which is a Partially Observable Markov Decision Process [37, 111] where the state is an access control state. An ACPOMDP extends an ACMMDP by considering a probability distribution $\pi : \Sigma \rightarrow [0, 1]$ over states and a set $\Theta$ of observations.

**Definition. 2** An Access Control Partially Observable Markov Decision Process is a tuple $(\Sigma, \mathcal{A}, \mathcal{P}, \Theta, \mathcal{C}, \mathcal{W})$ where $\Sigma = \mathcal{I} \times \mathcal{R}$ is a set of access control states, $\mathcal{A}$ is a set of decisions, $\mathcal{P} : \Sigma \times \mathcal{A} \times \Sigma \rightarrow [0, 1]$ is the probability function, such that $\mathcal{P}(\sigma_i, a, \sigma_j)$ stands for the probability of reaching the state $\sigma_j$ by executing the decision $a$ from the state $\sigma_i$, $\Theta$ is a set of observations, $\mathcal{C} : \mathcal{A} \times \Theta \times \Sigma \rightarrow [0, 1]$ is the observation model, such that $\mathcal{C}(a, \theta, \sigma_j)$ stands for the probability that we observe $\theta$ when we are in state $\sigma_j$, and our last decision was $a$, and $\mathcal{W} : \Sigma \times \mathcal{A} \times \Sigma \rightarrow \mathcal{U}$ is the reward function, such that $\mathcal{U}$ is a utility domain and $\mathcal{W}(\sigma_i, a, \sigma_j)$ stands for the reward associated with executing the decision $a$ from the state $\sigma_i$ and arriving
in the state \(\sigma_j\). When no confusion can arise, we write \(p_{ij}^\theta\), \(c_j^\theta\) and \(w_{ij}^\theta\) for \(P(\sigma_i,a,\sigma_j), C(a,\theta,\sigma_j)\) and \(W(\sigma_i,a,\sigma_j)\), respectively.

The optimal policy can therefore be calculated by:

\[
\delta^*(\pi) = \arg\max_{a \in A} \sum_{i,j} p_{ij}^\theta w_{ij}^\theta + \beta \sum_{i,j,\theta} p_{ij}^\theta c_j^\theta V^\delta(T(\pi, a, \theta)) \tag{5.3}
\]

Hence, the novelty of using (Partially Observable) Markov Decision Processes lies in the fact that the policy is inferred from the model, i.e., from the reward and the probability functions, rather than defined as a static structure. In other words, we believe that field experts should define and tune each individual parameter, in order to reflect in the best possible way the concrete security problem.

5.2.3 Implementation with GLPK/GMPL

We define in [100] an implementation of an ACMDP using GNU Linear Programming Kit (GLPK) which is a piece of software intended for solving large-scale linear programming problems, and which supports the GMPL language for modeling problems, a subset of AMPL. More specifically, we have shown how to instantiate an ACMDP from the perspective of a security engineer, who would be responsible for implementing a security mechanism for the following problem (loosely inspired from the healthcare environment):

1. There are two users, Alice and Bob, such that Alice (for instance, a physician) is more qualified than Bob (for instance, a nurse);
2. There are two resources, high and low, such that the resource high (for instance, one of Alice’s patient record) is more sensitive than low (for instance, the notice of usage of a drug);
3. Bob is normally not qualified enough to access high;
4. in case of emergency (for instance, the patient is having a heart attack), the resource high should be accessed.

In order to model this problem, we define the security information as a pair \((e,c)\), where \(e\) is a boolean indicating whether there is a current emergency and \(c\) is the set of current accesses, such that an access is simply a pair \((u,a)\), where \(u\) is a user and \(a\) a resource. In addition, we model a request directly as an access. In other words, if we write \(S\) for the set of users, \(O\) for the set of objects, we define \(I = B \times P(\Sigma \times O)\) and \(\Sigma = I \times (((\Sigma \times O) \cup \{e\})\), where \(e\) stands for the empty request. Hence, a state is a triple \((e,c,r)\), where \(r\) is either a pair \((u,a)\) or \(e\).

The reward function is defined using two sub-functions, as described in Table 5.2, where \(r_{ew_1}\) represents the reward for a single access, while \(r_{ew_2}\) represents the reward for a resource that is not accessed, according to whether there is an emergency or not. Given two states \(\sigma_i = (e,c,r)\) and \(\sigma_j = (e’,c’,r’),\) and an action \(a\), we define:

\[
\begin{align*}
w_{ij}^a &= \begin{cases} 0 & \text{if } r = e \\ \sum_{\{a|\exists a’ (u,o) \in c’\}} r_{ew_2}(a,e’) + r_{ew_1}(s,o) + \sum_{\{a’|\exists a’’ (u’’,o’) \in c’’\}} r_{ew_2}(a,e’’) & \text{if } a = \text{deny} \\ r_{ew_2}(a,e’) & \text{if } r = (s,o) \text{ and } a = \text{allow} \end{cases}
\end{align*}
\]

The probability transition function is defined such that the emergency status can switch from false to true with a probability \(p_e\), and such that the requested access is added if, and only if the decision is allow. Furthermore, we describe three possible behaviors in order to know the next request: unique indicates that there is only one request to evaluate (i.e., the next request is always \(e\)); once indicates that each request can only be asked once; all indicates that all requests are possible.

Table 5.3 presents the values associated with the request for each access, considering that it is the first request to control, and according to each possible request behavior and whether the state is in an emergency status or not. For instance, the first row of the table can be read as: if each request is unique,
Table 5.2: Reward functions

(a) Single access reward
\[ \text{rew}_1(u, o) \]

<table>
<thead>
<tr>
<th></th>
<th>Bob</th>
<th>Alice</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>high</td>
<td>-10</td>
<td>10</td>
</tr>
</tbody>
</table>

(b) Reward for resource non accessed
\[ \text{rew}_2(e, o) \]

<table>
<thead>
<tr>
<th></th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>high</td>
<td>0</td>
<td>-20</td>
</tr>
</tbody>
</table>

Table 5.3: Decision values for \( \beta = 0.9 \) and emergency probability of \( p_e = 0.1 \)

<table>
<thead>
<tr>
<th>Request</th>
<th>Emergency</th>
<th>Decision</th>
<th>(Alice, low)</th>
<th>(Alice, high)</th>
<th>(Bob, low)</th>
<th>(Bob, high)</th>
</tr>
</thead>
<tbody>
<tr>
<td>unique</td>
<td>false</td>
<td>deny</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td></td>
<td>true</td>
<td>allow</td>
<td>4</td>
<td>10</td>
<td>2</td>
<td>-10</td>
</tr>
<tr>
<td></td>
<td>true</td>
<td>deny</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
</tr>
<tr>
<td></td>
<td>true</td>
<td>allow</td>
<td>-14</td>
<td>10</td>
<td>-16</td>
<td>-10</td>
</tr>
<tr>
<td>once</td>
<td>false</td>
<td>deny</td>
<td>2.63</td>
<td>2.63</td>
<td>2.63</td>
<td>2.63</td>
</tr>
<tr>
<td></td>
<td>true</td>
<td>deny</td>
<td>-23.54</td>
<td>-23.54</td>
<td>-23.54</td>
<td>-23.54</td>
</tr>
<tr>
<td></td>
<td>true</td>
<td>allow</td>
<td>-15.54</td>
<td>14.15</td>
<td>-16.70</td>
<td>-1.59</td>
</tr>
<tr>
<td>all</td>
<td>false</td>
<td>deny</td>
<td>34.80</td>
<td>34.80</td>
<td>34.80</td>
<td>34.80</td>
</tr>
<tr>
<td></td>
<td></td>
<td>allow</td>
<td>40.80</td>
<td>55</td>
<td>38.80</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>true</td>
<td>deny</td>
<td>4.55</td>
<td>4.55</td>
<td>4.55</td>
<td>4.55</td>
</tr>
<tr>
<td></td>
<td></td>
<td>allow</td>
<td>10.55</td>
<td>55</td>
<td>8.55</td>
<td>35</td>
</tr>
</tbody>
</table>

and if the state is not in an emergency status, then denying the access (Alice, low) has a value of -2 while allowing it has a value of 4, thus meaning that the optimal policy is to allow this access in this situation.

Beyond the atomic values, an interesting point, fitting well with the problematics of quantitative access control, is for each access, to compare the difference between the values of each decision. For instance, in the last column of the fifth row, we can see denying or allowing the access (Bob, high) have very close values (34.80 against 35), which means that no decision is obviously the best, and it could require the assistance of a human expert to solve this case. On the other hand, in the last column of the sixth row, it is obvious that allowing the access (Bob, high) is the best decision.

We provide more experimental results in [100], by adjusting different parameters in order to observe the evolution of the values associated with each decision.

Contributions We have addressed the problem of quantitative access control by modeling an access control mechanism as a Markov Decision Process, extended this notion to Partially Observable Markov Decision Processes, and implemented our approach using linear programming through an example loosely inspired from the healthcare environment. An important aspect of our approach is to provide a metric for the value of security decisions, which is calculated as the potential impact of this decision. Hence, a security decision is no longer simply seen as a qualitative element, obtained from a static policy, but as a quantitative value computed from the environment, including several levels of uncertainty.

For further details on this work we refer the reader to our publications [92, 100].
6 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 2nd year deliverables of those work packages.

WP 2 – Integration of methodologies and tools in the WorkBench
Tools used in this Work Package have been integrated into the Service Development Environment (SDE) of WP 2, namely, UWE2XACML, XACML2FACPL, and MagicUWE (Section 2.2) as well as Avantssar-atse (CL-Atse) and the Avantssar Orchestrator (Section 2.3).

WP 7 – Secure service architectures and design
The work on “transformation verification” of Deliverable D7.3 [120] is closely related to WP 9. In that section, ATOS and INRIA present a prototype that performs automatic, unbounded verification of ATL transformations. This means checking that pre-condition and post-condition constraints, known to hold for the source model of the transformation, still hold for the transformed model. In other words, they verify that the transformation preserves these constraints. They embed the translations into statements of first-order logic and use SMT solvers such as Z3 and Yices to decide such problems.

WP 8 – Programming environments for secure and composable services
As already mentioned in the introduction, secure programming is covered in Deliverable D8.3 [117]. In particular, the section entitled “Programming Language Support” in that deliverable presents an advanced source code verification method that is supported by the VeriFast tool (which works on C and Java code). It takes as input a number of source files, annotated with method contracts written in separation logic, inductive data type and fixpoint definitions, lemma functions, and proof steps. The verifier checks that the program does not perform illegal operations such as dividing by zero or illegal memory accesses and that the assumptions described in method and contracts hold in each execution.

The contributions “Metric-aware secure service orchestration” and “Automatic quantification of cache side-channels” in Deliverable D8.3 are clearly related to quantitative security, the topic discussed in Section 5 of this deliverable.

WP 10 – Risk and cost aware SDLC
WP 10 is most closely related to quantitative security, that is, Section 5 of the present deliverable. Feedback from assurance methods may serve as input for risk and cost analysis. Conversely, risk analysis in early stages of the SDLC can identify possible vulnerabilities and attacks that can be used to set the focus for the testing phase (e.g., for the selection of test cases).

WP 11 – Future Internet application scenarios
WP 11 provides two case studies, which involve a representative mix of Future Internet features. One case study is in the area of e-Health and the other on smart grids. The methods and tools of several contributions presented in this deliverable have been validated on one of these case studies. The results are reported in Deliverable D11.3 [118]. In the following, we describe these links in more detail.

- The work on the analysis of communicating authorization policies reported in Section 2.3.2 includes a case study from the e-health domain, which is described in a section of Deliverable D11.3 entitled “Verification of workflows with revocation and delegation”.

- The contribution in Section 4.1.1 on non-interference for UML state charts has lead to a corresponding case study in the area of smart grids. It is documented in the section of D11.3 entitled “Non-Inference methods for a smart grid scenario”.

NESSoS - 256980 55
• The section entitled “Autodelegation for healthcare” in D11.3 applies the quantitative access control methods described in Section 5.2 to a e-Health case study.

• The D11.3 section entitled “Verification of an ASLaN++ Smart Grid Model”, although not directly related to a particular contribution of the present deliverable, applies the AVANTSSAR verification tool [11] to a detailed model of a smart grid scenario.
7 Conclusions

In this deliverable, we have reported on further development of our methods and on initial prototypal support for security assurance for services that we have obtained during the second year of the NESSoS project. We therefore take the occasion to briefly reflect on what we have achieved.

The work done so far in WP 9 covers the large majority of the tasks and activities set out in the NESSoS Description of Work. We record very good overall progress on all of the topics addressed, both in terms of the methods and of the tools developed. We can even report major advancements of the state-of-the-art in several areas, in particular in algorithmic verification, runtime monitoring, testing, debugging, and non-interference verification. Almost all methods and techniques are accompanied by a prototypal tool implementation or an extension of an existing tool.

In the following, we give a brief assessment of our results related to the different assurance activities and the corresponding WP 9 tasks and subtasks. We consider progress in terms of methods, tools, and the challenges set out in the conclusions of the state-of-the-art Deliverable ID9.1 [116]. First, we briefly recall these challenges:

C1 – Expressiveness More expressive specification languages and analysis methods supporting them are needed to address the challenges of Future Internet systems and services.

C2 – Distribution This challenge refers to the extension of methods and tools for the analysis of individual services or components to a distributed setting involving communication (e.g., distributed authorization policies, runtime monitoring of distributed applications).

C3 – Linking abstraction levels / SDLC phases The challenge here is to link the different abstraction levels along the SDLC in a semantically meaningful and sound way (e.g., model-based testing relates test cases extracted from design models to implementations).

C4 – Modularization / Compositionality This is needed to master the complexity of large systems. The challenge is to infer security of a composed system from the security of its components.

Here is a (non-exhaustive) description of our progress in the different activities.

**Refinement** We propose methods for doing both classical and non-interference refinement of security-sensitive systems including security protocols. Two methods and related tools based on Isabelle/HOL and Rodin/Event-B have emerged from this work. Challenges addressed: C1, C3.

**Algorithmic verification** We have significantly extended the expressiveness of existing formalisms in order to handle service-oriented architectures. These extensions include the handling of multiple intruders (C1, C2), XML rewriting attacks (C1), the integration of formalisms for distributed authorization policies and security protocols (tool-supported; C1, C2), and the handling of negative constraints in security protocol analysis, enabling more flexible orchestrator synthesis for service composition (CL-Atse tool; C1, C4).

**Model-extraction for formal analysis** We have translated UWE models into UML models for reachability checking in web applications using the Hugo/RT model checker (tool-supported; C1,C3).

We have also proposed a tool-chain that translates security requirements formulated in the high-level, graphical modeling language UWE to XACML policies and, in a further step, towards a formally founded language called FACPL (tool chain; C3).

Moreover, we have presented a method to formally prove two properties of ActionGUI models: invariant preservation and security awareness. We use OCL2FOL to map the relevant OCL expressions into first-order logic formulas, which we check using SMT solvers (partial tool support; C3).

**Testing and debugging** We have proposed a method for mutation-based testing of security protocols (jMuHLP-SL tool; C2, C3), a method for the comparison of XACML policy testing strategies (X-CREATE tool; C1), and a method to test the PDP implementation of an authorization system that supports usage control (tool-supported; C1).
We have also introduced two novel debugging techniques. First, differential slicing is a debugging technique that identifies causal execution differences. It has been applied to crash and malware analysis (tool-supported). Second, we propose early detection, a novel approach for finding and diagnosing use-after-free and double-free dangling pointer vulnerabilities (Undangle tool).

**Runtime verification** We have analyzed the impact of different time models on monitoring, tackled the problem of monitoring the usage of data in concurrent distributed systems (C2), and presented a policy language and monitoring algorithm that accounts for possibly incomplete and disagreeing logs (C1). On a more theoretical side, we have generalized Schneider’s work on enforceable security policies based on a practically relevant distinction between controllable and observable system actions when monitoring executions. We have also implemented a monitoring tool for compliance checking (MONOPOLY tool), where policies are specified as formulas of an expressive safety fragment of metric first-order temporal logic (C1).

Moreover, we have presented an enhanced authorization framework that is able to deal with long lasting accesses, preventing the continued usage of resources when the access rights have expired. We have extended the XACML policy schema with usage control scenarios and enhanced the XACML reference architecture with run-time enforcement of usage control. (tool-supported; C1)

**Bridging model-based and language-based security** We have presented an efficient verification strategy for non-trivial subset of UML state-charts that is sound with respect to classical non-interference as used in language-based security. Our technique, which extends previous work in the area, is fully automatic and can help to narrow the gap between theory and practice for information-flow in secure-software development in industrial context. We also present sufficient conditions for deciding the security of composed services from already verified components. (tool-supported; C1, C4)

**Quantitative security** We have presented a formal model for security metrics, established formal relations between different security metrics, and a generic method for the assessment of the security of complex services under different metrics (C1, C4).

We have addressed the problem of quantitative access control by modeling an access control mechanism as a Markov Decision Process, extended this notion to Partially Observable Markov Decision Processes, and implemented our approach using linear programming through an example loosely inspired by the healthcare environment (tool-supported; C1).

Moreover, we have created and compared two vulnerability prediction models in the domain of Android applications. The prediction models are based on machine-learning techniques.

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 assurance activities listed above. Here are some directions that deserve further attention:

- (C1) further expansion of the expressiveness boundaries for system modeling and property specification languages (e.g., web browser modeling, stronger attacker models, equivalence verification),
- (C2) the testing of entire distributed applications and the associated coordination, configuration, and data processing problems,
- (C3) establish a closer relation between model-based and language-based security,
- (C4) compositional / modular refinement and verification, which are notoriously difficult to achieve for security properties.
8 NESSoS WP 9 Second-year Publications


Bibliography


