

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 1 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

	Title: Concepts for Model-Based Security Testing	
	Version: 1.0 Date : 26.09.2011 Pages : 134	
	Editor: Wissam Mallouli	
	Reviewers: Bruno Legeard, Christian Wieser, Fredrik Seehusen	
To: DIAMONDS Consortium		
The DIAMONDS Consortium consists of: Codenomicon, Conformiq, Dornier Consulting, Ericsson, Fraunhofer FOKUS, FSCOM, Gemalto, Giesecke & Devrient, Grenoble INP, IT SudParis, itrust, Metso, Montimage, Norse Solutions, SINTEF, Smartesting, Secure Business Applications, Testing Technologies, Thales, Trusted Labs, TU Graz, University Oulu, VTT.		
Status: ☐ Draft ☐ To be reviewed ☐ Proposal ☒ Final / Released	Confidentiality: ☒ Public Intended for public use ☐ Restricted Intended for DIAMONDS consortium only ☐ Confidential Intended for individual partner only	

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 2 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

Deliverable ID: D2.WP2

Title:

Concepts for Model-Based Security Testing

Summary / Contents:

This document describes concepts dedicated to model-based security testing and used within the DIAMONDS project.

Contributors to the document:

Bernhard Aichernig (TU Graz), Michael Berger (Fraunhofer FOKUS), Julien Botella (Smartesting), Fabrice Bouquet (Smartesting), Ana Cavalli (IT SudParis), Fabien Duchène (Grenoble INP), Jürgen Großmann (Fraunhofer FOKUS), Roland Groz (Grenoble INP), Carlo Harpes (itrust), Paul Rascagnères (itrust) Andreas Hoffmann (Fraunhofer FOKUS), Ami Juuso (Codonomicon), Rauli Kaksonen (Codonomicon), Juha Koivisto (VTT), Felipe Lalanne (IT SudParis), Bruno Legeard (Smartesting), Stéphane Maag (IT SudParis), Wissam Mallouli (Montimage), Florian Marienfeld (Fraunhofer FOKUS), Nadja Menz (Fraunhofer FOKUS), Edgardo Montes de Oca (Montimage), Laurent Mounier (Grenoble INP), Sanjay Rawat (Grenoble INP), Jean-Luc Richier (Grenoble INP), Ina Schieferdecker (Fraunhofer FOKUS), Martin Schneider (Fraunhofer FOKUS), Fredrik Seehusen (SINTEF), Ari Takanen (Codonomicon), Juha Matti Tirila (Codonomicon), Tuomo Untinen (Codonomicon), Miia Vuontisjarvi (Codonomicon), Alain-Georges Vouffo Feudjio (Fraunhofer FOKUS), Bachar Wehbi (Montimage), Franz Wotawa (TU Graz).

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 3 of 134 Version: 1.0 Date : 26.09.2011 Status : Final Confid : Public
---	--	--

TABLE OF CONTENTS

Introduction	8
1. Monitoring and Inspection Concepts.....	9
1.1 Introduction	9
1.2 Security Properties Definition.....	9
1.2.1 UML Sequence Diagrams.....	10
1.2.2 State Machines	20
1.2.3 Specifying Transformations Using Sequence Diagrams	24
1.2.4 From sequence diagrams to state machines.....	25
1.3 Network Instrumentation and Data Extraction	38
1.3.1 Physical Level (L1)	38
1.3.2 Data Link (L2)	39
1.3.3 Transport Layer (L3).....	39
1.4 Advanced Network Traffic Analysis	40
1.4.1 MMT-Security Properties	40
1.4.2 Deep Packet/Flow Inspection	45
1.4.3 Network Traffic Analysis Based on MMT-Security Properties	45
1.5 Horn Logic Based Security Analysis – A Data Centric Approach	48
1.5.1 Protocol Messages and Traces	48
1.5.2 Horn Logic Syntax and Semantics.....	49
1.5.3 Security Properties Evaluation Algorithm on Execution Traces	51
1.5.4 Examples and Experiments	52
1.6 Machine Learning.....	54
1.6.1 Introduction	54
1.6.2 Challenges	54
1.6.3 Research.....	56
1.7 Binary Code Instrumentation	56
1.7.1 Introduction	56
1.7.2 Tracing.....	56
1.7.3 Debugging.....	62
1.8 Summary.....	65
2. Active Testing Concepts.....	66
2.1 Introduction	66
2.2 Model-Based Security Testing from Test Purposes	66
2.3 Integration of basic security rules involving atomic actions	69
2.3.1 Prohibition integration: $F(\text{start}(A) O^{[<]-d}\text{done}(B))$	69
2.3.2 Permission integration: $P(\text{start}(A) O^{[<]-d}\text{done}(B))$	71
2.3.3 Obligation integration.....	71
2.4 Combining Model-Recovery and Evolutionary Fuzzing	72
2.4.1 The General Approach	72
2.4.2 Smart Black-Box Fuzzing	73
2.4.3 Smart White-Box Fuzzing	76
2.5 Mutation Testing.....	78
2.5.1 Mutation Analysis Process.....	79
2.5.2 Mutation Operators	79
2.5.3 Mutations for Test Generator.....	80
2.6 Model-Based Mutation Testing	81
2.6.1 Introduction	81
2.6.2 A Car Alarm System	82
2.6.3 Mutation Killing Strategies	84
2.6.4 Test Case Execution Results for the Car Alarm System.....	86

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 4 of 134 Version: 1.0 Date : 26.09.2011 Status : Final Confid : Public
---	--	--

2.6.5	A Second Experiment: a Wheel Loader	87
2.6.6	Discussion.....	90
2.7	UMLsec	92
2.7.1	Running Example: Three-Tier Web Application	92
2.7.2	From CORAS Risk Analysis to UMLsec Models	94
2.7.3	Generating Test Cases from the Enriched System Model	95
2.7.4	Using UMLsec Stereotypes for Security Testing	96
2.7.5	Exploiting UMLsec Enriched State Charts to Generate Test Cases	96
2.7.6	Test Case Selection and Prioritization	99
2.7.7	Mapping to Other Case Studies: Giesecke und Devrient.....	101
2.8	Fuzzing UML Sequence Diagrams	101
2.8.1	Introduction	101
2.8.2	Related Work	102
2.8.3	Advantages of UML Sequence Diagrams when Fuzzing Behaviour.....	103
2.8.4	General Approach.....	104
2.8.5	Realization	105
2.8.6	Example: Application of fuzzing operations to a UML sequence diagrams	110
2.8.7	Concatenation of basic fuzzing operations	113
2.9	Conclusion	116
3.	Risk Analysis for Risk Based Testing.....	117
3.1	Mirroring Risk Analysis Techniques to Telecom Use Case.....	117
3.1.1	Brief Overview of the Current Risk Assessment Process	117
3.1.2	Mirroring Techniques to Telecom Use Case	118
3.2	Tracing Between Risk Modelling Elements and Test Artifacts	118
3.2.1	Overview on Risk Analysis Concept.....	118
3.2.2	A Minimal Set of Requirements Analysis and Architectural Design Concepts.....	120
3.2.3	Overview on (Model Based) Testing Concepts	120
3.2.4	Risk-Based Security Testing.....	123
3.2.5	Traceability for Risk-Based Testing	125
3.3	Test Case Selection	127
3.4	Summary	128
	Glossary.....	129
	References.....	130

FIGURES

Figure 1:	Example of a state machine.....	20
Figure 2:	State machines W and W' are obtained by transformation without and with condition Last, respectively	29
Figure 3:	Machines A and B are nondeterministic while C is deterministic.....	29
Figure 4:	State machine P and its inversion P'	30
Figure 5:	State machine Q and its inversion Q'	31
Figure 6:	State machine W and its inversion.....	33
Figure 7:	State machine W, its incorrect inversion $ph_2(W) = W'$, and its correct inversion W''	34
Figure 8:	State machine W and its (incorrect) inversion W' and (correct) inversion W''	35
Figure 9:	IP duplication	43
Figure 10:	Tracing schema.....	57
Figure 11:	Example of <code>strace</code> output on Linux.....	57
Figure 12:	API Monitor interface on Windows	58
Figure 13:	DTrace example	58
Figure 14:	SystemTrap example	58
Figure 15 :	RegMon interface on Windows	59

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 5 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

Figure 16: TCPView output on Windows.....	60
Figure 17: Simple <code>getpid()</code> example	61
Figure 18: Custom <code>getpid()</code> library.....	61
Figure 19: Custom <code>getpid()</code> library V2	61
Figure 20: Example of buffer overflow.....	62
Figure 21: <code>dummy()</code> function disassembled	63
Figure 22: Immunity debugger at <code>dummy()</code> function.....	63
Figure 23: Immunity debugger show how the application crashed	64
Figure 24: Application crashed	64
Figure 25: Core dump generation	64
Figure 26: Core dump analysis	65
Figure 27: Process of model-based security testing with test purposes	67
Figure 28: Test purpose example and unfolding	68
Figure 29: Transition decomposition	70
Figure 30: Prohibition rule integration: $F(start(A) \mid O^{<d} done(B))$	70
Figure 31: Obligation Rule Integration.....	72
Figure 29: A general view of the proposed approach	72
Figure 30: Our proposed approach on combining model inference and fuzzing for automated vulnerability search on a black-box SUT.....	74
Figure 31: Test interface of the car alarm system.....	83
Figure 32: State machine of the car alarm system.....	83
Figure 33: The computation steps of the conformance checker Ulysses.	84
Figure 34: Topology of a three-tier Web application.	92
Figure 35: SQL injection risk analysis.	93
Figure 36: Cross site scripting risk analysis.	93
Figure 37: UMLsec stereotypes [67].	95
Figure 38: Simple state machine	97
Figure 39: State machine (high operations and states are red)	97
Figure 40: State machine mutated for security testing	98
Figure 41: Simple CORAS risk model containing two threats, one vulnerability, one threat scenario and one unwanted incident	99
Figure 42: Simplified banking example	111
Figure 43: Invalid sequence diagram obtained by moving a message	112
Figure 44: Invalid sequence diagram obtained by negating an interaction constraint	112
Figure 45: Invalid sequence diagram obtained by changing the bounds of a loop combined fragment.....	113
Figure 46: Fuzzed Sequence Diagram by Applying Three Different Fuzzing Operations	114
Figure 47: <i>Alternatives</i> Combined Fragment with Fuzzed Second Interaction Operand.....	115
Figure 48: Modified <i>Alternatives</i> Combined Fragment Ensuring Only Invalid Sequences Can Be Generated	116
Figure 49: The model-based testing elements [81].....	122
Figure 50: Information model for testing concepts	123
Figure 51: Risk-based testing	124

TABLES

Table 1: Number of test cases for the CAS generated by our eight different approaches.	84
Table 2: Injected faults in the CAS implementation per method.	86
Table 3: Overview of how many faulty SUTs of the CAS survived the generated test cases.....	87
Table 4: Number of test cases for both wheel loader models generated by strategy S5.	88
Table 5: Number of random test cases for both wheel loader models (S7).....	88
Table 6: Number of test cases for both models of the wheel leader generated by strategy S6.	89
Table 7: Killing rates of generated test cases per strategy.	90

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 6 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Table 8: Basic Fuzzing Operations	110
Table 9: CORAS main threat analysis concepts (see [48])	119
Table 10: CORAS main risk estimation concepts (see [48])	119
Table 11: A minimal set of requirements analysis and architectural design concepts.....	120
Table 12: Overview on (model based) testing concepts	122
Table 13: Relations between risk assessment concepts and test identification concepts	125
Table 14: Relations between risk assessment concepts and test selection concepts.....	126
Table 15: Relations between risk assessment concepts and testing concepts for risk control	126

HISTORY

Vers.	Date	Author	Description
0.1	2011/09/26	W. Mallouli	Document creation
0.1	2011/10/28	S. Maag	Section 1.5 added
0.1	2011/10/28	J-L. Richier	Section 2.4 added
0.1	2011/10/28	J.Grossmann	Sections 2.6, 2.7, 3.2 added
0.1	2011/10/30	B. Legeard	Section 2.2 added
0.1	2011/11/01	C. Harpes	Section 1.7 added
0.1	2011/11/01	W. Mallouli	Section 1.4 added
0.1	2011/11/03	F. Wotawa	Section 2.5 added
0.1	2011/11/04	J. Koivisto	Section 3.3 added
0.1	2011/11/04	R. Groz	Section 2.4 updated
0.1	2011/11/11	F. Marienfeld	Section 2.6 updated
0.1	2011/11/14	P. Pietikainen	Section 1.3 added
0.1	2011/11/21	F. Seehusen	Section 1.2 added
0.1	2011/11/22	W. Mallouli	Missing sections added
0.2	2011/11/23	W. Mallouli	Reviewable document creation
1.0	2011/12/07	W. Mallouli	Final document creation + some comments need to be addressed
1.0	2011/12/12	W. Mallouli	Final version of D2.WP3

APPLICABLE DOCUMENT LIST

Ref.	Title, author, source, date, status	DIAMONDS ID
1		

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 7 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

EXECUTIVE SUMMARY

DIAMONDS considers the particular issue of security testing of networked systems to validate the dependability of networked systems in face of malice, attack, error or mischance. Testing is still the main method to reliably check the functionality, robustness, performance, scalability, reliability and resilience of systems as it is the only method to derive objectively characteristics of a system in its target environment. A number of approaches have long been around to target specific attacks on systems (e.g. vulnerability scanners), but when we refer here to the more systematic testing of systems with respect to specified policies or security properties, testing a system for its security is a relatively new concern that has started to be addressed in the last few years.

D1.WP2 already reviewed the state-of-the-arts methods used in security testing. This document D2.WP2 describes the main concepts dedicated to model-based security testing used by the different project partners. It can be considered as a progress report of the work done of each partner to develop its testing approach.

Monitoring (referred also by passive testing) consists of detecting faults in a system under test by observing its input/output behaviours without interfering with its normal operations. The usual approach of passive testing consists of recording the trace produced by the implementation under test and comparing this trace with a specification. Other approaches explore relevant properties required for a correct implementation, and then check them on the implementation traces of the system under test. Chapter 1 describes the concepts followed by DIAMONDS partners in the monitoring and inspection field.

Active testing is accomplished by applying a sequence of inputs to the implementation, by means of an external tester, and verifying whether the sequence of outputs is the one specified. These sequences may be constructed from formal models. Most of the works are on testing software control parts, and technologies are related to the approaches of model-based testing. Traditional testing methods tend to test a system as a whole or to test their components in isolation. Testing these systems as whole becomes difficult due to the large number of combinations of system states and variable values, known as the state space explosion. It is a challenge to be able to minimize the number of tests needed while guaranteeing good fault coverage. This standard method is mainly oriented towards practical needs. Chapter 2 describes partners' approaches for active testing.

Security testing often lacks systematic approaches, that enable the efficient and goal oriented identification, selection and execution of test cases. A successful approach to resolve this problem is to use risk-oriented testing, where one uses software risks analysis as the guiding factor to solve decision problems during testing, e.g. the selection and prioritization of test cases. This approach is advocated by different security testing standards and manuals (NIST, OSSTMM etc.). Chapter 3 describes the partners approaches in this domain.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 8 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

INTRODUCTION

Testing is still the main method to check the functionality, robustness, performance, scalability, reliability and resilience of systems as it is the only method to derive objectively characteristics of a system in its target environment.

In the case of security, software systems are examined, using software security testing, for security properties such as confidentiality, integrity, authentication, authorization, availability, and non-repudiation. In general the software security testing activities can be divided into functional security testing and security vulnerability testing [74]. While security functional testing is used to check the functionality, efficiency and availability of the specified and carefully planned security functionalities and systems (e.g. firewalls, authentication and authorization subsystems, access control), security vulnerability testing directly addresses the identification and discovery of actually undiscovered system vulnerabilities that are introduced by security design flaws.

Systematically testing a system for its security is a relatively new concern. Of course, a number of tools have long been around to target specific attacks on systems (e.g. vulnerability scanners). Different broader approaches have also been proposed in the recent years.

This document lists a number of approaches dedicated to testing software security and followed by DIAMONDS partners. It is organized as follows:

Chapter 1 presents monitoring and inspection concepts mainly in network security and binary code instrumentation contexts. Monitoring (also referred to passive testing) is the activity of detecting faults in a system under test by observing its input/output behaviours without interfering with its normal operations.

In Chapter 2, active testing concepts are presented; active testing consists in applying a sequence of inputs to the implementation, by means of an external tester, and verifying whether the sequence of outputs is the one expected.

Chapter 3 describes risk based testing concepts. Risk-oriented testing or risk-based testing characterize a methodology that makes software risks the guiding factor to solve decision problems during testing, e.g. the selection and prioritization of test cases.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 9 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

1. MONITORING AND INSPECTION CONCEPTS

1.1 INTRODUCTION

Network monitoring is a laborious and demanding task that is vital for the network infrastructure. Service providers are constantly striving to keep the network operation stable, smooth and safe. If the network becomes vulnerable, under attack or breaks down, even for a small period of time, the service provider's ability to deliver secure and high-quality services would be compromised. Network administrators must be proactive rather than reactive: monitoring the network traffic and security and performance at all times, and verifying that security threats do not occur within the network perimeter.

In this section, we describe the work done by DIAMONDS partners in the context of monitoring and inspection concepts:

- SINTEF presents in section 1.2 how security properties can be defined based on UML sequence diagrams and state machines and how we can make transformation between these two models
- OUSPG describes in section 1.3 their methodology for network instrumentation and data extraction in different network ISO layers.
- Montimage presents in section 1.4 its advances network traffic analysis based on MMT-Security properties.
- Institut Telecom SudParis contribution is presented in section 1.5 where they describe horn logic based security analysis of captured trace files.
- Machine learning techniques are presented by VVT in section 1.6
- And finally, Ittrust present their vision to Binary code instrumentation in section 1.7.

1.2 SECURITY PROPERTIES DEFINITION

Model based testing is based on the idea of specifying a functional model M of the system under test (SUT) which describes what the SUT is supposed to do. The SUT adheres to the model M if every execution trace of the model is admissible by the SUT . To test that SUT adheres to M , we generate execution traces (called *tests*) from M , and execute these on the SUT to see if they are admissible. Usually, M describes an infinite number of executions, so the tests have to be selected according to some coverage criteria, for instance that all transitions of the model (if the model is expressed as a state machine) must be covered by the selected executions.

It is often convenient to specify the model using actions/events that are on a higher level of abstraction than the actions/events of the SUT . To test whether the SUT adheres to such a model, one has to transform the execution traces of the abstract model into execution traces of the SUT before the tests are executed. In addition, when testing security properties, it is often more convenient to specify a negative model M' describing what SUT is *not* supposed to do than to specify a function model describing what SUT is supposed to do. To test that the SUT adheres to a negative model M' using conventional model based testing techniques, M' has to be *inverted* into a functional model M admitting exactly those execution traces that are not admitted by M' .

In this section, we present a method for specifying security policies (we use the term policy instead of model to emphasise that we are focusing on security) using UML sequence diagrams describing what the SUT is *not* supposed to do. To take into account that actions/events of the policies may be on a higher level of abstraction than the action/events of the SUT , we also describe how UML sequence diagrams can be used in order to specify transformations from high-level events to low-level behaviour.

In contrast to model based testing, monitoring is based on the idea of recording the execution of SUT at runtime, and checking whether the execution is admissible by the model M . Hence, the SUT adheres to the

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 10 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

model M if every execution of SUT is admissible by M . In this section, we will consider this notion of adherence. In particular, we describe an approach for transforming security policies expressed as UML sequence diagrams into state machines governing the behaviour of monitoring mechanisms.

First, in section 1.2.1 and section 1.2.2, we describe the syntax and the semantics of UML sequence diagrams and UML state machines, respectively. Then, in section 1.2.4, we describe how sequence diagrams can be used to specify transformations from high-level actions to lower-level behaviour. Finally, in section 1.2.4 we describe a transformation from UML sequence diagrams into state machines that govern the behaviour of monitoring mechanisms.

1.2.1 UML Sequence Diagrams

In this section, we first present the syntax and semantics of UML sequence diagrams. Then, we define what it means for a system to adhere to a sequence diagram policy.

1.2.1.1 Syntax

We use the following syntactic categories to define the textual representation of sequence diagrams:

$$\begin{aligned}
 ax &\in \mathbf{AExp} && \text{arithmetic expressions} \\
 bx &\in \mathbf{BExp} && \text{boolean expressions} \\
 sx &\in \mathbf{SExp} && \text{string expressions}
 \end{aligned}$$

We let **Exp** denote the set of all arithmetic, Boolean, and string expressions, and we let ex range over this set. We denote the empty expression by ϵ . We let **Val** denote the set of all values, i.e., numerals, strings, and Booleans (**t** or **f**) and we let **Var** denote the set of all variables. Obviously, we have that **Val** $\subset$ **Exp** and **Var** $\subset$ **Exp**.

Every sequence diagram is built by composing atoms or sub-diagrams. The atoms of a sequence diagram are the *events*, *constraints*, and the *assignments*. These constructs are presented in following section, while the syntax of sequence diagrams in general and their semantics is presented in the next sections.

Events, constraints, assignments

The atoms of a sequence diagram are the *events*, *constraints*, and the *assignments*. An event is a pair (k, m) of a kind k and a message m . An event of the form $(!, m)$ represents a transmission of message m , whereas an event of the form $(?, m)$ represents a reception of m . We let **E** denote the set of all events:

$$\mathbf{E} \stackrel{\text{def}}{=} \{!, ?\} \times \mathbf{M} \quad (1)$$

where **M** denotes the set of all messages.

On events, we define a kind function $k._ \in \mathbf{E} \rightarrow \{!, ?\}$ and a message function $m._ \in \mathbf{E} \rightarrow \mathbf{M}$:

$$k._(k, m) \stackrel{\text{def}}{=} k \quad m._(k, m) \stackrel{\text{def}}{=} m \quad (2)$$

Messages are of the form (l, l_r, si) where l represents the transmitter lifeline of the message, l_r represents the receiver lifeline of the message, and si represents the signal of the message. We let **L** denote the set of all lifelines, and **SI** denote the set of all signals. The set **M** of all messages is then defined by

$$\mathbf{M} \stackrel{\text{def}}{=} \mathbf{L} \times \mathbf{L} \times \mathbf{SI} \quad (3)$$

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 11 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

On messages, we define a transmitter function $tr_ \in \mathbf{M} \rightarrow \mathbf{L}$ and a receiver function $re_ \in \mathbf{M} \rightarrow \mathbf{L}$:

$$tr.(l_t, l_r, si) \stackrel{\text{def}}{=} l_t \quad re.(l_t, l_r, si) \stackrel{\text{def}}{=} l_r \quad (4)$$

We let the transmitter and receiver functions also range over events, $tr_ , re_ \in \mathbf{E} \rightarrow \mathbf{L}$:

$$tr.(k, m) \stackrel{\text{def}}{=} tr.m \quad re.(k, m) \stackrel{\text{def}}{=} re.m \quad (5)$$

We define a lifeline function $l_ \in \mathbf{E} \rightarrow \mathbf{L}$ that returns the lifeline of an event and a function $l^{-1} _ \in \mathbf{E} \rightarrow \mathbf{L}$ that returns the inverse lifeline of an event (i.e., the receiver of its message if its kind is transmit and the transmitter of its message if its kind is receive):

$$l.e \stackrel{\text{def}}{=} \begin{matrix} tr.e & \text{if } k.e = ! \\ re.e & \text{if } k.e = ? \end{matrix} \quad l^{-1}.e \stackrel{\text{def}}{=} \begin{matrix} tr.e & \text{if } k.e = ? \\ re.e & \text{if } k.e = ! \end{matrix} \quad (6)$$

A signal is a tuple $(nm, ex_1, \dots, ex_n)$ where nm denotes the signal name, and $ex_1, \dots, ex_n$ are the parameters of the signal. We usually write $nm(ex_1, \dots, ex_n)$ instead of $(nm, ex_1, \dots, ex_n)$. Formally, the set of all signals is defined

$$\mathbf{SI} \stackrel{\text{def}}{=} \mathbf{Nm} \times \mathbf{Exp}^* \quad (7)$$

where A^* yields the set of all sequences over the elements in the set A .

A signal may contain special so-called *parameter variables* that are bound to values upon the occurrence of the signal. Parameter variables are similar to free normal variables (normal variables that have not explicitly been assigned to a value). However, they differ in that parameter variables contained in a loop will be assigned to new values for each iteration of the loop.

A parameter variable is a pair (vn, i) consisting of variable name vn and an index i (this is a natural number). When a sequence diagram is executed, the index of a parameter variable contained in a loop will be incremented by one for each iteration of the loop. This is to ensure that the parameter variable is given a new value when the loop is iterated. Hence, the index of a parameter variable is only used for bookkeeping purposes during execution, and it will never be explicitly specified in a graphical diagram.

In a graphical sequence diagram, parameter variables are distinguished from normal variables by writing the parameter variables in boldface. The index of a parameter variable in a graphical sequence diagram is always initially assumed to be zero.

The set of all parameter variables $\mathbf{PVar}$ is defined

$$\mathbf{PVar} \stackrel{\text{def}}{=} \mathbf{VN} \times \mathbf{N} \quad (8)$$

where $\mathbf{VN}$ is the set of all variable names and $\mathbf{N}$ is the set of all natural numbers. We assume that

$$\mathbf{PVar} \subset \mathbf{Var} \quad (9)$$

A *constraint* is an expression of the form

$$\text{constr}(bx, l)$$

where bx is a boolean expression and l is a lifeline. Intuitively, interactions occurring after a constraint in a diagram will only take place if and only if the boolean expression of the constraint evaluates to true. We denote the set of all constraints by $\mathbf{C}$ and we let c range over this set.

An *assignment* is an expression of the form

$$\text{assign}(x, ex, l)$$

where x is a normal variable, i.e. $x \in \mathbf{Var} \setminus \mathbf{PVar}$, ex is an expression, and l is a lifeline. Intuitively, the assignment represents the binding of expression ex to variable x on lifeline l . We let $\mathbf{A}$ denote the set of all assignments and we let a range over this set.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 12 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

We define the function $l. \in \mathbf{A} \cup \mathbf{C} \rightarrow \mathbf{L}$ which yields the lifeline of an assignment or constraint as follows

$$l.\text{constr}(bx, l) \stackrel{\text{def}}{=} l \quad l.\text{assign}(x, ex, l) \stackrel{\text{def}}{=} l \quad (10)$$

We denote by $\mathbf{E}_l$, $\mathbf{C}_l$, and $\mathbf{A}_l$, the set of all events, constraints, and assignments with lifeline l , respectively, i.e.,

$$\mathbf{E}^l \stackrel{\text{def}}{=} \{e \in \mathbf{E} \mid l.e = l\} \quad \mathbf{C}^l \stackrel{\text{def}}{=} \{c \in \mathbf{C} \mid l.c = l\} \quad \mathbf{A}^l \stackrel{\text{def}}{=} \{a \in \mathbf{A} \mid l.a = l\} \quad (11)$$

Diagrams

In the previous section, we presented the atomic constructs of a sequence diagram. In this section, we present the syntax of sequence diagrams in general.

Definition 1 (Sequence diagram) Let e , bx , l , x , and ex denote events, boolean expressions, lifelines, variables, and expressions, respectively. The set of all syntactically correct sequence diagram expressions $\mathbf{D}$ is defined by the following grammar:

$$d ::= \text{skip} \mid e \mid \text{constr}(bx, l) \mid \text{assign}(x, ex, l) \mid \text{refuse}(d) \mid \text{loop}\langle 0..* \rangle(d) \mid d_1 \text{ seq } d_2 \mid d_1 \text{ alt } d_2 \mid d_1 \text{ par } d_2$$

The base cases implies that any event (e), skip, constraint ($\text{constr}(bx, l)$), or assignment ($\text{assign}(x, ex, l)$) is a sequence diagram. Any other sequence diagram is constructed from the basic ones through the application of operators for negation ($\text{refuse}(d)$), iteration ($\text{loop}\langle 0..* \rangle(d)$), weak sequencing ($d_1 \text{ seq } d_2$), choice ($d_1 \text{ alt } d_2$), and parallel execution ($d_1 \text{ par } d_2$).

We define some functions over the syntax of diagrams. We let the function $eca._ \in \mathbf{D} \rightarrow \mathbf{P}(\mathbf{E} \cup \mathbf{C} \cup \mathbf{A})$ return all events, constraints, and assignments present in a diagram. The function is defined as follows

$$\begin{aligned} eca.\alpha &\stackrel{\text{def}}{=} \{\alpha\} && \text{for } \alpha \in \mathbf{E} \cup \mathbf{C} \cup \mathbf{A} \\ eca.\text{skip} &\stackrel{\text{def}}{=} \emptyset \\ eca.(\text{op}(d)) &\stackrel{\text{def}}{=} eca.d && \text{for } \text{op} \in \{\text{refuse}, \text{loop}\langle 0..* \rangle\} \\ eca.(d_1 \text{ op } d_2) &\stackrel{\text{def}}{=} eca.d_1 \cup eca.d_2 && \text{for } \text{op} \in \{\text{seq}, \text{alt}, \text{par}\} \end{aligned} \quad (12)$$

Note that we henceforth let α denote an arbitrary event, constraint, or assignment, i.e., $\alpha \in \mathbf{E} \cup \mathbf{C} \cup \mathbf{A}$.

The function $ll._ \in \mathbf{D} \rightarrow \mathbf{P}(\mathbf{L})$ returns all lifelines of a diagram:

$$ll.d \stackrel{\text{def}}{=} \bigcup_{\alpha \in eca.d} \{l.\alpha\} \quad (13)$$

We denote by $\mathbf{D}_l$, the set of all diagrams with only one lifeline l , i.e.,

$$\mathbf{D}^l \stackrel{\text{def}}{=} \{d \in \mathbf{D} \mid ll.d = \{l\}\} \quad (14)$$

The function $msg._ \in \mathbf{D} \rightarrow \mathbf{P}(\mathbf{M})$ returns all the messages of a diagram:

$$msg.d \stackrel{\text{def}}{=} \bigcup_{e \in (eca.d \cap \mathbf{E})} \{m.e\} \quad (15)$$

The projection operator $\pi_-(_) \in \mathbf{L} \times \mathbf{D} \rightarrow \mathbf{D}$ that projects a diagram to a lifeline is defined

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 13 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

$$\begin{aligned}
\pi_l(\alpha) &\stackrel{\text{def}}{=} \alpha && \text{if } l.\alpha = l \\
\pi_l(\alpha) &\stackrel{\text{def}}{=} \text{skip} && \text{if } l.\alpha \neq l \\
\pi_l(\text{skip}) &\stackrel{\text{def}}{=} \text{skip} \\
\pi_l(\text{op } d) &\stackrel{\text{def}}{=} \text{op}(\pi_l(d)) && \text{for } \text{op} \in \{\text{refuse}, \text{loop}(0..*)\} \\
\pi_l(d_1 \text{ op } d_2) &\stackrel{\text{def}}{=} \pi_l(d_1) \text{ op } \pi_l(d_2) && \text{for } \text{op} \in \{\text{seq}, \text{alt}, \text{par}\}
\end{aligned} \tag{16}$$

We let $\text{var} \in (\mathbf{Exp} \cup \mathbf{M}) \rightarrow \mathbf{P}(\mathbf{Var})$ be the function that yields the variables in an expression or the variables in the arguments of a signal of a message. We lift the function to diagrams as follows

$$\text{var}(d) \stackrel{\text{def}}{=} \bigcup_{m \in \text{msg}.d} \text{var}(m) \cup \bigcup_{\text{constr}(bx, l) \in \text{eca}.d \cap \mathbf{C}} \text{var}(bx) \cup \bigcup_{\text{assign}(x, ex, l) \in \text{eca}.d \cap \mathbf{A}} (\{x\} \cup \text{var}(ex)) \tag{17}$$

Syntactic constraints

We impose some restrictions on the set of syntactically correct sequence diagrams **D**. We describe four rules which are taken from [48]. First, we assert that a given event should syntactically occur only once in a diagram. Second, if both transmitter and the receiver lifelines of a message are present in a diagram, then both the transmit event and the receive event of that message must be in the diagram. Third, if both the transmit event and the receive event of a message are present in a diagram, then they have to be inside the same argument of the same high level operator. The constraint means that in the graphical notion, messages are not allowed to cross the frame of a high level operator or the dividing line between the arguments of a high level operator. Fourth, the operator refuse is not allowed to be empty, i.e., to contain only the skip diagram.

The four rules described above are formally defined in [48]. These rules ensure that the operational semantics is sound and complete with the denotational semantics of sequence diagrams as defined in [48]. In this report, we define ten additional rules and we say that a diagram d is *well formed* if it satisfies these:

SD1 The variables of the lifelines of d are disjoint.

SD2 All parameter variables of d have index 0.

SD3 If m is a message in d , then the arguments of the signal of m must be distinct parameter variables only.

SD4 The first atomic construct of each lifeline in d must be an assignment (not a constraint or an event).

SD5 All parameter variables that occur inside a loop in d do not occur outside that loop.

SD6 All loops in d must contain at least one event.

SD7 No two events in d contain the same parameter variables.

SD8 For each lifeline in d , each constraint c must be followed by an event e (not an assignment or a constraint). In addition, the parameter variables of c must be a subset of the parameter variables of e .

SD9 For each lifeline in d , the parameter variables of an assignment must be a subset of parameter variables of each event that proceeds it on the lifeline. If the assignment has no proceeding events on the lifeline, then the assignment cannot contain parameter variables.

SD10 All variables in d (except for the parameter variables) must explicitly be assigned to a value before they are used.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 14 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

The purpose of the syntax constraints is to ensure that the sequence diagram can be correctly transformed into a state machine.

Note that any graphical sequence diagram can be described by a textual diagram that satisfies conditions **SD1 - SD4**.

To obtain a diagram that satisfies **SD1** and **SD2** we have to rename variables on each lifeline and set the index of all parameter variables to zero. To obtain a diagram that satisfies condition **SD3** we convert arguments (that are not parameter variables) of the signal of an event into constraints proceeding the event. For instance, the diagram

$$(!, (l_t, l_r, msg(ex)))$$

– which does not satisfy **SD3** because ex might not be a parameter variable – can be converted into the diagram

$$constr(px=ex, l_t) seq (!, (l_t, l_r, msg(px))) \quad \text{for some } px \in PVar$$

which does satisfy **SD3**. Here $px=ex$ is a boolean expression that yields true if and only if px is equal to ex .

If a sequence diagram d does not satisfy condition **SD4**, then a dummy assignment can be added to start of each lifeline in d that assigns some value to a variable that is not used in d .

1.2.1.2 Semantics

In this section, we define the operational semantics of UML sequence diagrams based on the semantics defined in [48]. The operational semantics tells us how a sequence diagram is executed step by step. It is defined as the combination of two labeled transition systems, called the *execution system* and the *projection system*.

These two systems work together in such a way that for each step in the execution, the projection system updates the execution system by selecting an enabled event to execute and returning the state of the diagram after the execution of the event.

The projection system

The projection system is used for finding enabled events at each step of execution. The projection system (as well as the execution system) is formally described by a labeled transition system (LTS).

Definition 2 (Labeled transition system (LTS)) *A labeled transition system over the set of labels LE is a pair $(Q, \mathcal{R})$ consisting of*

- *a (possibly infinite) set Q of states;*
- *a ternary relation of $\mathcal{R} \subseteq (Q \times LE \times Q)$, known as a transition relation.*

We usually write $q \xrightarrow{le} q' \in (Q, \mathcal{R})$ if $(q, le, q') \in \mathcal{R}$, or just $q \xrightarrow{le} q'$ if $(Q, \mathcal{R})$ is clear from the context. If $s = \langle le_1, le_2, \dots, le_n \rangle$, we write $q \xrightarrow{s} q'$ for $q \xrightarrow{le_1} q_1 \xrightarrow{le_2} q_2 \dots \xrightarrow{le_n} q'$. For the empty sequence $\langle \rangle$, we write $q \xrightarrow{\langle \rangle} q'$ iff $q = q'$.

To define the projection system, we make use of a notion of structural congruence which defines simple rules under which sequence diagrams should be regarded as equivalent.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 15 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Definition 3 (Structural congruence) *Structural congruence over sequence diagrams, written $\equiv$, is the congruence over $\mathbf{D}$ determined by the following equations:*

1. $d \text{ seq skip} \equiv d, \text{ skip seq } d \equiv d$
2. $d \text{ par skip} \equiv d, \text{ skip par } d \equiv d$
3. $\text{skip alt skip} \equiv \text{skip}$
4. $\text{loop}(0..*)(\text{skip}) \equiv \text{skip}$

The projection system is an LTS whose states are pairs $\Pi(L, d)$ consisting of a set of lifelines L and a diagram d . If the projection system has a transition from $\Pi(L, d)$ to $\Pi(L, d_-)$ that is labeled by, say event e , then we understand that e is enabled in diagram d , and that d_- is obtained from d by removing event e . Whenever the high level construct alt, refuse, or loop is enabled in a diagram, the projection system will produce a so-called silent event that indicates the kind of construct that has been executed. For instance, each state of the form $\Pi(L, \text{refuse}(d))$ has a transition to $\Pi(L, d)$ that is labeled by the silent event τ_{refuse} .

The set of lifelines L that appears in the states of the projection system is used to define the transition rules of the weak sequencing operator **seq**. The weak sequencing operator defines a partial order on the events in a diagram; events are ordered on each lifeline and ordered by causality, but all other ordering of events is arbitrary. Because of this, there may be enabled events in both the left and the right argument of a **seq** if there are lifelines present in the right argument of the operator that are not present in the left argument. The set of lifelines L is used to keep track of which lifelines are shared by the arguments of **seq**, and which lifelines only occur in the right argument (but not the left) of **seq**.

The following definition of the projection system is based on [48].

Definition 4 (Projection system) *The projection system is an LTS over*

$$\alpha_T \in \{\tau_{\text{refuse}}, \tau_{\text{alt}}, \tau_{\text{loop}}\} \cup \mathbf{E} \cup \mathbf{C} \cup \mathbf{A}$$

whose states are

$$\Pi(., .) \in \mathbb{P}(\mathbf{L}) \times \mathbf{D}$$

and whose transitions are exactly those that can be derived by the following rules

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 16 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

$$\begin{array}{c}
\frac{}{\Pi(L, \alpha) \xrightarrow{\alpha} \Pi(L, \text{skip})} \text{if } l.\alpha \in L \quad \frac{}{\Pi(L, \text{refuse}(d)) \xrightarrow{\tau_{\text{refuse}}} \Pi(L, d)} \text{if } ll.d \cap L \neq \emptyset \\
\\
\frac{}{\Pi(L, d_1 \text{ alt } d_2) \xrightarrow{\tau_{\text{alt}}} \Pi(L, d_i)} \text{if } ll.(d_1 \text{ alt } d_2) \cap L \neq \emptyset \text{ for } i \in \{1, 2\} \\
\\
\frac{\Pi(ll.d_1 \cap L, d_1) \xrightarrow{\alpha_r} \Pi(ll.d_1 \cap L, d'_1)}{\Pi(L, d_1 \text{ seq } d_2) \xrightarrow{\alpha_r} \Pi(L, d'_1 \text{ seq } d_2)} \text{if } ll.d_1 \cap L \neq \emptyset \\
\\
\frac{\Pi(L \setminus ll.d_1, d_2) \xrightarrow{\alpha_r} \Pi(L \setminus ll.d_1, d'_2)}{\Pi(L, d_1 \text{ seq } d_2) \xrightarrow{\alpha_r} \Pi(L, d_1 \text{ seq } d'_2)} \text{if } L \setminus ll.d_1 \neq \emptyset \\
\\
\frac{\Pi(L, d_1) \xrightarrow{\alpha_r} \Pi(L, d'_1)}{\Pi(L, d_2) \xrightarrow{\alpha_r} \Pi(L, d'_2)} \text{if } d_1 \equiv d_2 \text{ and } d'_1 \equiv d'_2 \\
\\
\frac{\Pi(ll.d_1 \cap L, d_1) \xrightarrow{\alpha_r} \Pi(ll.d_1 \cap L, d'_1) \quad \Pi(ll.d_2 \cap L, d_2) \xrightarrow{\alpha_r} \Pi(ll.d_2 \cap L, d'_2)}{\Pi(L, d_1 \text{ par } d_2) \xrightarrow{\alpha_r} \Pi(L, d'_1 \text{ par } d_2) \quad \Pi(L, d_1 \text{ par } d_2) \xrightarrow{\alpha_r} \Pi(L, d_1 \text{ par } d'_2)} \\
\\
\frac{}{\Pi(L, \text{loop}\langle 0..* \rangle(d)) \xrightarrow{\tau_{\text{loop}}} \Pi(L, \text{skip alt}(d \text{ seq loop}\langle 0..* \rangle(d)))} \text{if } ll.d \cap L \neq \emptyset
\end{array}$$

For more explanation of the rules of the projection system, the reader is referred to [48].

The projection system of Def. 4 is based on [48] where parameter variables are not taken into consideration. Recall that each parameter variable is bound to a new value upon the occurrence of the event it is contained in. This has the consequence that parameter variables occurring inside a loop are bound to new values for each iteration of the loop. Thus to modify the projection system of Def. 4 to take this into account, we only need to modify the rule for $\text{loop}\langle 0..* \rangle$ (the last rule of Def. 4). To simulate the fact that parameter variables are bound to new values in each iteration of the loop, we let the projection system rename all parameter variables by incrementing their index for each iteration of the loop. Formally, we make use of the function $ipv(_) \in \mathbf{PVar} \rightarrow \mathbf{PVar}$ that increments the index of a parameter variable by one, i.e.,

$$ipv((vn, i)) = (vn, i + 1)$$

The function is lifted to diagrams such that $ipv(d)$ yields the diagram obtained from d by incrementing all its parameter variables by one. The revised projection system is now given by the following definition.

Definition 5 (Revised projection system) *The revised projection system that handles parameter variables is the LTS over*

$$\alpha_r \in \{\tau_{\text{refuse}}, \tau_{\text{alt}}, \tau_{\text{loop}}\} \cup \mathbf{E} \cup \mathbf{C} \cup \mathbf{A}$$

whose states are

$$\Pi'(_, _) \in \mathbb{P}(\mathbf{L}) \times \mathbf{D}$$

and whose transitions are exactly those that can be derived by the rules of Def. 4 except for rule for $\text{loop}\langle 0..* \rangle$ which is redefined as follows:

$$\frac{}{\Pi'(L, \text{loop}\langle 0..* \rangle(d)) \xrightarrow{\tau_{\text{loop}}} \Pi'(L, \text{skip alt}(d \text{ seq loop}\langle 0..* \rangle(ipv(d))))} \text{if } ll.d \cap L \neq \emptyset$$

Evaluation and data states

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 17 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

In order to define the operational semantics of sequence diagrams, we need to describe how the data states change throughout execution. In this section, we present some auxiliary functions that are needed for this purpose.

An expression $ex \in \mathbf{Exp}$ is closed if $\text{var}(ex) = \emptyset$. We let $\mathbf{CExp}$ denote the set of closed expressions, defined as:

$$\mathbf{CExp} \stackrel{\text{def}}{=} \{ex \in \mathbf{Exp} \mid \text{var}(ex) = \emptyset\}$$

We assume the existence of a function $\text{eval} \in \mathbf{CExp} \rightarrow \mathbf{ValU}\{\perp\}$ that evaluates any closed expression to its value. If an expression ex is not well formed or otherwise cannot be evaluated (e.g., because of division by zero), then $\text{eval}(ex) = \perp$. The evaluation function is lifted to signals, messages, and events such that $\text{eval}(si)$, $\text{eval}(m)$, $\text{eval}(e)$ evaluate all expressions of signal si , message m , and event e , respectively. For example, we have that

$$\text{eval}(\text{msg}(1 + 2, 4 - 1)) = \text{msg}(\text{eval}(1 + 2), \text{eval}(4 - 1)) = \text{msg}(3, 3)$$

If an expression ex in signal si is not well formed, i.e., $\text{eval}(ex) = \perp$, then $\text{eval}(si) = \perp$. If e is an event (k, m) and si the signal of m , then we also have that $\text{eval}(m) = \perp$ and $\text{eval}(e) = \perp$.

Let $\sigma \in \mathbf{Var} \rightarrow \mathbf{Exp}$ be a mapping from variables to expressions. We denote such a mapping $\sigma = \{x_1 \mapsto ex_1, x_2 \mapsto ex_2, \dots, x_n \mapsto ex_n\}$ for distinct $x_1, x_2, \dots, x_n \in \mathbf{Var}$ and for $ex_1, ex_2, \dots, ex_n \in \mathbf{Exp}$. If $ex_1, ex_2, \dots, ex_n \in \mathbf{Val}$ we call it a data state. We let Σ denote the set of all mappings and $\hat{\Sigma}$ denote the set of all data states. We use the same convention for the set of all events $\mathbf{E}$, and denote by $\hat{\mathbf{E}}$, the set of all events whose signals have only values as arguments.

The empty mapping is denoted by $\emptyset$. $\text{Dom}(\sigma)$ denotes the domain of σ , i.e.,

$$\text{Dom}(\{x_1 \mapsto ex_1, x_2 \mapsto ex_2, \dots, x_n \mapsto ex_n\}) \stackrel{\text{def}}{=} \{x_1, x_2, \dots, x_n\}$$

We let $\sigma[x \mapsto ex]$ denote the mapping σ except that it maps x to ex , i.e.,

$$\begin{aligned} \{x_1 \mapsto ex_1, \dots, x_n \mapsto ex_n\}[x \mapsto ex] &\stackrel{\text{def}}{=} \begin{cases} \{x_1 \mapsto ex_1, \dots, x_n \mapsto ex_n, x \mapsto ex\} & \text{if } x \neq x_i \text{ for all } i \in \{1, \dots, n\} \\ \{x_1 \mapsto ex_1, \dots, x_i \mapsto ex, \dots, x_n \mapsto ex_n\} & \text{if } x = x_i \text{ for some } i \in \{1, \dots, n\} \end{cases} \end{aligned}$$

We generalize $\sigma[x \mapsto ex]$ to $\sigma[\sigma']$ in the following way:

$$\sigma[\{x_1 \mapsto ex_1, \dots, x_n \mapsto ex_n\}] \stackrel{\text{def}}{=} \sigma[x_1 \mapsto ex_1] \cdots [x_n \mapsto ex_n]$$

The mapping is lifted to expressions such that $\sigma(ex)$ yields the expression obtained from ex by simultaneously substituting the variables of ex with the expressions that these variables map to in σ . For example, we have that $\{y \mapsto 1, z \mapsto 2\}(y + z) = 1 + 2$. We furthermore lift σ to signals, messages, and events such that $\sigma(si)$, $\sigma(m)$, and $\sigma(e)$ yields the signal, message, and event obtained from si , m , and e , respectively, by substituting the variables of their expressions according to σ .

Execution system and trace semantics for sequence diagrams

The execution system of the operational semantics tells us how to execute a sequence diagram in a step by step manner. Unlike the projection system, the execution system keeps track of the *communication medium* and *data states* in addition to the diagram state. Thus a state of the execution system is a triple consisting of a communication medium, diagram, and data state:

$$\mathbf{AXS} \stackrel{\text{def}}{=} \mathbf{B} \times \mathbf{D} \times \hat{\Sigma}_T$$

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 18 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Here $\hat{\Sigma}_T$ denotes the set of *total* data states, i.e., the set of all data states σ satisfying

$$\text{Dom}(\sigma) = \text{Var}$$

We assume a communication model where each message has its own channel from the transmitter to the receiver, something that allows for message overtaking. The communication medium keeps track of messages that are sent between lifelines of a diagram, i.e., the messages of transmission events are put into the communication medium, while the messages of receive events are removed from the communication medium.

It is only necessary to keep track of the communication between those lifelines that are present in a sequence diagram; messages received from the environment (i.e., from lifelines not present in a diagram) are always assumed to be enabled.

The states of the communication medium are of the form (M, L) where M is a set of messages and L is a set of lifelines under consideration, i.e., the lifelines that are not part of the environment. The set of all communication medium states $\mathbf{B}$ is defined by

$$\mathbf{B} \stackrel{\text{def}}{=} \mathbb{P}(\mathbf{M}) \times \mathbb{P}(\mathbf{L}) \quad (18)$$

We define two functions for manipulating the communication medium: $\text{add}, \text{rm} \in \mathbf{B} \times \mathbf{M} \rightarrow \mathbf{B}$. The function $\text{add}(\beta, m)$ adds the message m to the communication medium β , while $\text{rm}(\beta, m)$ removes the message m from the communication medium β . We also define the predicate $\text{ready} \in \mathbf{B} \times \mathbf{M} \rightarrow \text{Bool}$ that for a communication medium β and a message m yields true if β is in a state where it can deliver m , and false otherwise. Formally, we have

$$\begin{aligned} \text{add}((M, L), m) &\stackrel{\text{def}}{=} (M \cup \{m\}, L) \\ \text{rm}((M, L), m) &\stackrel{\text{def}}{=} (M \setminus \{m\}, L) \\ \text{ready}((M, L), m) &\stackrel{\text{def}}{=} \text{tr}.m \notin L \vee m \in M \end{aligned} \quad (19)$$

We are now ready to define the execution system for sequence diagrams.

Definition 6 (Execution system) *The execution system is an LTS whose states are*

$$\mathbf{B} \times \mathbf{D} \times \hat{\Sigma}_T$$

whose labels are

$$\{\tau_{\text{refuse}}, \tau_{\text{alt}}, \tau_{\text{loop}}, \tau_{\text{assign}}, \mathbf{t}, \mathbf{f}, \perp\} \cup \mathbf{E}$$

and whose transitions are exactly those that can be derived from the following rules

$$\frac{\Pi'(ll.d, d) \xrightarrow{\tau} \Pi'(ll.d, d')}{[\beta, d, \sigma] \xrightarrow{\tau} [\beta, d', \sigma]} \text{ for } \tau \in \{\tau_{\text{refuse}}, \tau_{\text{loop}}, \tau_{\text{alt}}\}$$

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 19 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

$$\begin{array}{c}
\frac{\Pi'(ll.d, d) \xrightarrow{(!, m)} \Pi'(ll.d, d')}{[\beta, d, \sigma] \xrightarrow{(!, eval(\sigma(m)))} [add(\beta, eval(\sigma(m))), d', \sigma]} \\
\frac{\Pi'(ll.d, d) \xrightarrow{(? , m)} \Pi'(ll.d, d')}{[\beta, d, \sigma] \xrightarrow{(? , m')} [rm(\beta, m'), d', \sigma]} \text{ if } ready(\beta, m') \wedge eval(\sigma(m)) = m' \\
\frac{\Pi'(ll.d, d) \xrightarrow{assign(x, ex, l)} \Pi'(ll.d, d')}{[\beta, d, \sigma] \xrightarrow{\tau_{assign}} [\beta, d', \sigma[x \mapsto eval(\sigma(ex))]]} \\
\frac{\Pi'(ll.d, d) \xrightarrow{constr(bx, l)} \Pi'(ll.d, d')}{[\beta, d, \sigma] \xrightarrow{eval(\sigma(bx))} [\beta, d', \sigma]}
\end{array}$$

See [48] for more details on the rules of the execution system.

The trace semantics of a sequence diagram is a pair consisting of a positive trace set and a negative trace set. The traces of a diagram d are obtained by recording the events occurring on the transitions of the execution system when executing d until it is reduced to a skip (which means that the diagram cannot be further executed).

To distinguish negative from positive traces, we make use of the silent event $refuse$. That is, if a transition labeled by $refuse$ is taken during execution, then this means that a negative trace is being recorded. Otherwise the trace is positive.

Definition 7 (Trace semantics) *The trace semantics of d , written $\llbracket d \rrbracket$, is then defined by*

$$\begin{aligned}
\llbracket d \rrbracket \stackrel{\text{def}}{=} & (\{s \mid \mathbf{E} \in \hat{\mathbf{E}}^* \mid \\
& \exists \beta \in \mathbf{B} : \exists \sigma, \sigma' \in \hat{\Sigma}_T : \\
& [(\emptyset, ll.d), d, \sigma] \xrightarrow{s} [\beta, skip, \sigma'] \wedge s|_{\{\tau_{refuse}, f, \perp\}} = \langle \rangle\}, \\
& \{s \mid \mathbf{E} \in \hat{\mathbf{E}}^* \mid \\
& \exists \beta \in \mathbf{B} : \exists \sigma, \sigma' \in \hat{\Sigma}_T : \\
& [(\emptyset, ll.d), d, \sigma] \xrightarrow{s} [\beta, skip, \sigma'] \wedge s|_{\{\tau_{refuse}, f, \perp\}} \in \{\tau_{refuse}\}^+\})
\end{aligned}$$

Note that the projection function $/$ takes a set A and a sequence s and yields the sequence s/A obtained from s by removing all elements not in A . Note also that A_+ denotes the set of sequences of A with at least one element, i.e., $A_+ =_{\text{def}} A^* \setminus \{\langle \rangle\}$.

1.2.1.3 Policy adherence for sequence diagrams

In this section, we define what it means for a system to adhere to a policy expressed by a sequence diagram.

A system (interpreted as a set of traces of events) adheres to a sequence diagram policy if none of the traces of the system has a negative trace of a lifeline in the sequence diagram as a sub-trace. A trace $s = \langle e_1, \dots, e_n \rangle$ is a sub-trace of t , written $s \triangleleft t$, iff

$$s_1 \frown \langle e_1 \rangle \frown \dots \frown s_n \frown \langle e_n \rangle \frown s_{n+1} = t \quad (20)$$

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 20 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

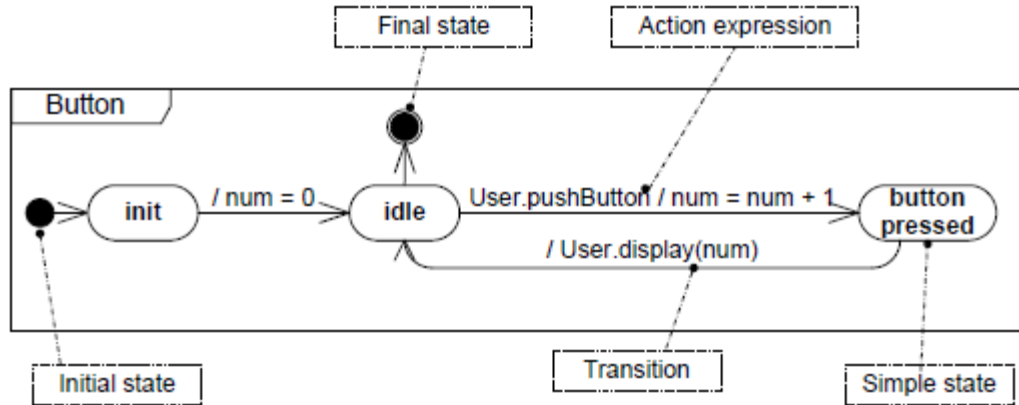


Figure 1: Example of a state machine

for some $s_1, \dots, s_{n+1} \in \mathbf{E}^*$. See [85] for a more precise definition.

We first formally define adherence for diagrams consisting of a single lifeline.

Definition 8 (Policy adherence of single lifeline sequence diagrams) *Let d be a single lifeline diagram, i.e., $d \in \mathbf{D}^l$ for some lifeline l , and let Φ denote the traces of a system. Then the system adheres to the policy d , written $d \rightarrow_{da} \Phi$, iff*

$$(s \in H_{neg} \wedge t \in \Phi|_{\mathbf{E}^l}) \implies \neg(s \triangleleft t) \quad \text{for } \llbracket d \rrbracket = (H_{pos}, H_{neg})$$

Note that the projection operator $/$ is lifted to sets of sequences such that Φ/A yields the set obtained from Φ by projecting each sequence of Φ to A , i.e., $\Phi/A = \{s/A \mid s \in \Phi\}$.

Adherence for general sequence diagrams (i.e., sequence diagrams that may contain more than one lifeline) is captured by the following definition.

Definition 9 (Policy adherence of sequence diagrams) *Let d be a sequence diagram, i.e., $d \in \mathbf{D}$ and let Φ denote the traces of a system. Then the system adheres to the policy d , written $d \rightarrow_{dag} \Phi$, iff*

$$\pi_l(d) \rightarrow_{da} \Phi \quad \text{for all } l \in ll.d$$

1.2.2 State Machines

In this section, we define the syntax and the semantics of UML inspired state machines. We also define what it means for a system to adhere to a policy expressed as a state machine.

1.2.2.1 Syntax

As illustrated in Figure 1, the constructs which are used for specifying state machines are *initial state*, *simple state*, *final state*, *transition*, and *action expression*.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 21 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

A state describes a period of time during the life of a state machine. The three kinds of states, *initial state*, *simple states*, and *final states*, are graphically represented by a black circle, a box with rounded edges, and a black circle encapsulated by another circle, respectively.

A *transition* represents a move from one state to another. In the graphical diagrams, transitions are labeled by *action expressions* of the form

$$nm.si[bx]/ef$$

Here the expression $nm.si$ where nm is a state machine name and si is a signal is called an *event trigger*. The expression $[bx]$ where bx is a boolean expression is called a *guard*, and ef is called an *effect*. Intuitively, the action should be understood as follows: when signal si is received from a state machine with name nm and the boolean expression bx evaluates to true, then the effect ef is executed. An effect is a sequence of assignments and/or an output expression of the form $nm.si$ representing the transmission of signal si to the state machine with name nm .

We will henceforth consider action expressions that contain at most one event. In our formal representation of state machines, we will therefore use action expressions of the form (e, bx, sa) where e is an input or output event, bx is a boolean expression (the guard) and sa is a sequence of assignments of the form $((x_1, ex_1), \dots, (x_n, ex_n))$. Formally, the set of all action expressions w.r.t. to the set of events E is defined by

$$\mathbf{Act}_E \stackrel{\text{def}}{=} (E \cup \{\epsilon\}) \times \mathbf{BExp} \times (\mathbf{Var} \times \mathbf{Exp})^*$$

Note that the event is optional in an action. An action without an event is of the form $(_, bx, sa)$. We will henceforth use e_ϵ to denote an arbitrary event or an empty expression, i.e., e_ϵ denotes a member of $\mathbf{E} \cup \{\epsilon\}$.

The alphabet of a state machine is a set of events containing signals whose arguments are distinct parameter variables. We require that all events in the alphabet are distinct when two events e and e' are considered equal if they have the same name and the same number of arguments.

To make this more precise, we let $\mathbf{E}_{pv}$ denote the set of all events whose signals contain distinct parameter variables only, i.e.,

$$\begin{aligned}
& \forall (k, (nm_t, nm_r, st(ex_1, \dots, ex_n))) \in \mathbf{E} : \\
& \quad \wedge ex_1 \in \mathbf{PVar} \wedge \dots \wedge ex_n \in \mathbf{PVar} \\
& \quad \wedge \forall i, j \in \{1, \dots, n\} : \\
& \quad \quad i \neq j \implies ex_i \neq ex_j \\
& \quad \iff (k, (nm_t, nm_r, st(ex_1, \dots, ex_n))) \in \mathbf{E}_{pv}
\end{aligned} \tag{21}$$

Note that the formula is written in a style suggested by Lamport [84]. Here, the arguments of a conjunction may be written as an aligned list where $\wedge$ is the first symbol before each argument. A similar convention is used for disjunctions. Also, indentation is sometimes used instead of parentheses.

We write $e = e'$ if events e and e' have the same kind, transmitter, and receiver and their signals have the same name and the same numbers of arguments, i.e.,

$$\begin{aligned}
& (k, (nm_t, nm_r, st(ex_1, \dots, ex_j))) = (k', (nm'_t, nm'_r, st'(ex'_1, \dots, ex'_k))) \\
& \iff k = k' \wedge nm_t = nm'_t \wedge nm_r = nm'_r \wedge st = st' \wedge j = k
\end{aligned} \tag{22}$$

We are now ready to define the syntax of state machines precisely.

Definition 10 (State machines) A state machine is a tuple $(\mathcal{E}, Q, \mathcal{R}, q_I, \mathcal{F})$ consisting of

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 22 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

- an alphabet $\mathcal{E} \subseteq \mathbf{E}_{pv}$ where $e, e' \in \mathcal{E} \implies \neg(e = e')$;
- a set of states Q ;
- a transition relation $\mathcal{R} \subseteq Q \times \mathbf{Act}_{\mathcal{E}} \times Q$;
- an initial state $q_I \in Q$;
- a set of final states $\mathcal{F} \subseteq Q$

The set of all state machines is denoted by **SM**.

We define the functions for obtaining the alphabet, states, transitions, initial state, and final states of a state machine:

$$\begin{aligned}
 \text{alph}((\mathcal{E}, Q, \mathcal{R}, q_I, \mathcal{F})) &\stackrel{\text{def}}{=} \mathcal{E} \\
 \text{states}((\mathcal{E}, Q, \mathcal{R}, q_I, \mathcal{F})) &\stackrel{\text{def}}{=} Q \\
 \text{trans}((\mathcal{E}, Q, \mathcal{R}, q_I, \mathcal{F})) &\stackrel{\text{def}}{=} \mathcal{R} \\
 \text{init}((\mathcal{E}, Q, \mathcal{R}, q_I, \mathcal{F})) &\stackrel{\text{def}}{=} q_I \\
 \text{final}((\mathcal{E}, Q, \mathcal{R}, q_I, \mathcal{F})) &\stackrel{\text{def}}{=} \mathcal{F}
 \end{aligned}$$

Syntax constraints

We impose one restriction on the set of syntactically correct action expressions $\mathbf{Act}_{\mathcal{E}}$; the parameter variables of the guard and the assignment sequence must be a subset of the parameter variables of the event (if the event is present in the action):

$$(e, bx, sa) \in \mathbf{Act}_{\mathcal{E}} \implies (pvar(bx) \cup pvar(sa)) \subseteq pvar(e) \quad (23)$$

where $pvar$ yields the set of all parameter variables in an expression, assignment sequence, or an event.

We define four syntax rules for state machines, and we say that a state machine SM is *well formed* if it satisfies these rules:

SM1 The initial state of SM has zero ingoing transitions.

SM2 The initial state of SM has exactly one outgoing transition, and the action expression of this transition does not contain an event or a guard.

SM3 Each transition of SM (except the initial transition) is labeled by an action expression that contains an event.

SM4 All variables (except parameter variables) in SM must be explicitly assigned to a value before they are used.

1.2.2.2 Semantics

In this section, we define the semantics of state machines. First we define the execution graph of a state machine, then we define the traces obtained by executing a state machine.

The execution graph of a state machine SM is an LTS whose states are pairs $[q, \sigma]$ where q is a state of SM and σ is a data state. The transitions of the execution graph are defined in terms of the transitions of SM . That is, if SM has a transition from q to q' that is labeled by (e, bx, sa) and the signal of e has no arguments, then the execution graph has a transition from $[q, \sigma]$ to $[q', \sigma]$ that is labeled by e provided that the guard bx

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 23 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

is evaluated to true under state σ . Here, the data state σ' is equal to σ except that the variables of sa are assigned to new values as specified by sa . To make this precise, we define the function

$$as2ds \in \hat{\Sigma} \times (\text{Var} \times \text{Exp})^* \rightarrow \hat{\Sigma}$$

which takes a data state σ and an assignment sequence sa and yields a new updated data state. Formally,

$$\begin{aligned} as2ds(\sigma, ()) &\stackrel{\text{def}}{=} \sigma \\ as2ds(\sigma, (x, ex) \frown sa) &\stackrel{\text{def}}{=} as2ds(\sigma[x \mapsto eval(\sigma(ex))], sa) \end{aligned} \quad (24)$$

If the signal of event e contains arguments, i.e., parameter variables, then these variables are bound the new arbitrary values. In this case, the guard bx and the event e are evaluated under some data state $\sigma[\sigma']$ where σ' is an arbitrary mapping whose domain equals the parameter variables of e .

Definition 11 (Execution graph of state machines) *The execution graph of state machine $SM = (\mathcal{E}, \mathcal{Q}, \mathcal{R}, q_I, \mathcal{F})$, written $EG(SM)$, is the LTS over $\{\epsilon\} \cup \mathbf{E}$ whose states are*

$$\mathcal{Q} \times \hat{\Sigma}_T$$

and whose transitions are exactly those that can be derived from the following rule

$$\frac{q \xrightarrow{(e_\epsilon, bx, sa)} q' \in \mathcal{R}}{[q, \sigma] \xrightarrow{eval(\sigma[\sigma'])(e_\epsilon)} [q', as2ds(\sigma[\sigma'], sa)]} \text{ if } eval(\sigma[\sigma'])(bx) = \tau \wedge Dom(\sigma') = pvar(e_\epsilon)$$

The trace semantics of a state machine is the set of sequences obtained by recording the events occurring in each path from the initial state to a final state of the state machine.

Definition 12 (Trace semantics of state machines) *The trace semantics of a state machine $SM = (\mathcal{E}, \mathcal{Q}, \mathcal{R}, q_I, \mathcal{F})$, written $\llbracket SM \rrbracket$, is defined by*

$$\begin{aligned} \llbracket SM \rrbracket &\stackrel{\text{def}}{=} \{s | \mathbf{E} \in \hat{\mathbf{E}}^* \mid \\ &\quad \exists q' \in \mathcal{F} : \exists \sigma, \sigma' \in \hat{\Sigma}_T : \\ &\quad [q_I, \sigma] \xrightarrow{s} [q', \sigma'] \in EG(SM)\} \end{aligned}$$

1.2.2.3 Policy adherence for state machines

In this section, we define what it means for a system to adhere to a policy expressed as a state machine. We also define what it means for a system to adhere to a set of state machines.

Intuitively, a system S adheres to a state machine policy SM if all execution traces of S (when restricted to the alphabet of SM) are described by SM . This is formally captured by the following definition.

Definition 13 (Policy adherence for a state machine) *Let SM be a state machine defining a policy and let Φ denote the traces of a system. Then the system adheres to SM , written $SM \rightarrow_{sa} \Phi$, iff*

$$\Phi|_E \subseteq \llbracket SM \rrbracket$$

where $E \stackrel{\text{def}}{=} \{e \in \hat{\mathbf{E}} \mid e' \in \text{alph}(SM) \wedge e = e'\}$.

Adherence for a set of state machines is precisely captured by the following definition.

Definition 14 (Policy adherence for a set of state machines) *Let SMS be a set of state machines and let Φ denote the traces of a system. Then the system adheres to SMS , written $SMS \rightarrow_{sag} \Phi$, iff*

$$SM \rightarrow_{sa} \Phi \quad \text{for all } SM \in SMS$$

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 24 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

1.2.3 Specifying Transformations Using Sequence Diagrams

In this section, we show how transformations can be expressed in terms of sequence diagrams.

1.2.3.1 Transformation specifications

A *transformation specification* is a set of *mapping rules*. A mapping rule is a pair (dp, dp') consisting of a left hand side *sequence diagram pattern* dp and a right hand side *sequence diagram pattern* dp' . A *sequence diagram pattern* is a sequence diagram whose atoms (events, constraints, and assignments) may contain *meta variables*.

We let **MVar** denote the set of all meta variables and we let mv range over this set. Events that may contain meta variables are called *event patterns*. The set **EP** of all event patterns is defined

$$\mathbf{EP} \stackrel{\text{def}}{=} \mathbf{K} \times ((\mathbf{L} \cup \mathbf{MVar}) \times (\mathbf{L} \cup \mathbf{MVar}) \times (\mathbf{SIP} \cup \mathbf{MVar}))$$

Here **SIP** denotes the set of all *signal patterns*. This set is defined by

$$\mathbf{SIP} \stackrel{\text{def}}{=} (\mathbf{NM} \cup \mathbf{MVar}) \times (\mathbf{ExpP})^*$$

where **ExpP** is an expression that may contain meta variables (in addition to normal variables and parameter variables).

A *constraint pattern* is an expression of the form

$$\text{constr}(bxp, l)$$

where bxp is a boolean expression that may contain meta variables. We let **CP** denote all constraint patterns.

An *assignment pattern* is an expression of the form

$$\text{assign}(x, exp, l)$$

where exp is an expression that may contain meta variables. We let **AP** denote the set of all assignment patterns.

Definition 15 (Sequence diagram pattern) *The set of sequence diagram patterns DP is defined by the following syntax*

$$dp ::= mv \mid ep \mid cp \mid ap \mid \text{refuse}(dp) \mid \text{loop}(0..*)(dp) \mid dp_1 \text{ seq } dp_2 \mid dp_1 \text{ alt } dp_2 \mid dp_1 \text{ par } dp_2$$

A sequence diagram pattern is either a meta variable (mv), an event pattern (ep), a constraint pattern (cp), an assignment pattern (ap), or the composition of one or more diagram patterns.

1.2.3.2 Transformations

In this section, we define the function induced by a transformation specification. Intuitively, when a transformation specification ts is applied to a sequence diagram d , all fragments of d that *match* a left hand side pattern of a mapping rule in ts are replaced by the right hand side pattern of the mapping rule.

Matching is defined in terms of a substitution $sub \in \mathbf{MVar} \rightarrow (\mathbf{D} \cup \mathbf{Exp})$ that replaces meta variables by diagrams or expressions. Any substitution sub is lifted to diagram patterns such that $sub(dp)$ yields the diagram obtained from dp by simultaneously replacing all meta variables in dp by diagrams or expressions according to sub . The set of all substitution is denoted by Sub .

A diagram pattern dp *matches* a diagram d if there is a substitution sub such that

$$sub(dp) = d$$

We say that the domain of a mapping rule (dp, dp') , written $Dom((dp, dp'))$, is the set of all diagrams that can be matched by its left hand side pattern, i.e.,

$$Dom((dp, dp')) \stackrel{\text{def}}{=} \{d \in \mathbf{D} \mid \exists sub \in Sub : sub(dp) = d\}$$

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 25 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

To ensure that transformation specifications induce functional transformations, we require that the mapping rules of a transformation specification must have disjoint domains, i.e., each transformation specification ts must satisfy the following constraint

$$\forall (dp_1, dp'_1) \in ts : \forall (dp_2, dp'_2) \in ts : \\ (dp_1, dp'_1) \neq (dp_2, dp'_2) \implies \text{Dom}((dp_1, dp'_1)) \cap \text{Dom}((dp_2, dp'_2)) = \emptyset$$

Definition 16 (Function induced by a transformation specification) *The function $T_{ts} \in \mathbf{D} \rightarrow \mathbf{D}$ induced by transformation specification ts is defined as follows*

```

if sub(dp) = d for some (dp, dp') ∈ ts and sub ∈ Sub
  then Tts(d) = sub(dp')
else if d = d1 op d2 for some d1, d2 ∈ D and op ∈ { seq, alt, par }
  then Tts(d) = Tts(d1) op Tts(d2)
else if d = op(d1) for some d1 ∈ D and op ∈ { loop(0..*), refuse }
  then Tts(d) = op(Tts(d1))
else
  Tts(d) = d

```

1.2.4 From sequence diagrams to state machines

In this section, we define the transformation from sequence diagrams to state machines. In general, the transformation of a sequence diagram yields a set of state machines, i.e., one state machine for each lifeline in the sequence diagram.

The main requirement to the transformation is that it should be adherence preserving.

Definition 17 (Adherence preservation) *Let $T \in \mathbf{D} \rightarrow \mathbb{P}(\mathbf{SM})$ be a transformation from sequence diagrams to sets of state machines. Then T is adherence preserving if for every system with traces Φ and sequence diagram policy d , the system adheres to d if and only if it adheres to $T(d)$, i.e.,*

$$d \rightarrow_{dag} \Phi \Leftrightarrow T(d) \rightarrow_{sag} \Phi$$

We first, in Sect. 1.2.4.1, define the transformation from a sequence diagram with only one lifeline to a state machine. Then, in Sect. 1.2.4.2, we define the transformation from (general) sequence diagrams to state machine sets. We show that this transformation is adherence preserving for policies that are composed of sub-policies with disjoint sets of variables.

1.2.4.1 From single lifeline diagrams to state machines

The transformation from a single lifeline diagram d to a state machine has two phases. In phase 1, the sequence diagram d is transformed into a state machine SM whose trace semantics equals the negative trace set of $[[d]]$. In phase 2, SM is inverted into the state machine SM' whose semantics is the set of all traces that do not have a trace of SM as a sub-trace.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 26 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Definition 18 (Single lifeline sequence diagram to basic state machine)

The transformation $d2p \in \mathbf{D} \rightarrow \mathbf{SM}$ from single lifeline diagrams to state machines is defined by

$$d2p \stackrel{\text{def}}{=} ph2 \circ ph1$$

where $ph1$ and $ph2$ represent phase 1 and 2 (as defined below).

Phase 1

In phase 1, the sequence diagram d is transformed into a state machine SM that describes the negative traces of d . The state machine SM corresponds to the projection system induced by d . That is, if the projection system has a transition $\Pi(ll.d, d) \xrightarrow{\epsilon} \Pi(ll.d, d')$, then SM has a transition $q \xrightarrow{\epsilon} q'$ where q and q' correspond to $\Pi(ll.d, d)$ and $\Pi(ll.d, d')$, respectively. However, some transitions of the projection system are truncated. In particular,

- all silent events are removed, e.g., if $\Pi(ll.d, d) \xrightarrow{\tau_{alt}} \Pi(ll.d, d') \xrightarrow{\epsilon} \Pi(ll.d, d'')$, then SM has a transition $q \xrightarrow{\epsilon} q''$;
- constraints are concatenated with succeeding events and assignments concatenated with preceding events, e.g., if $\Pi(ll.d, d) \xrightarrow{\text{constr}(bx, l)} \Pi(ll.d, d_1) \xrightarrow{\epsilon} \Pi(ll.d, d_2) \xrightarrow{\text{assign}(x, ex, l)} \Pi(ll.d, d_3)$, then SM has a transition $q \xrightarrow{(e, bx, ((x, ex)))} q_3$

To define this precisely, we introduce the notion of experiment relation.

Definition 19 (Experiment relations) The relations $\Rightarrow$, $\overset{\alpha}{\Rightarrow}$, and $\overset{s}{\Rightarrow}$ for any $\alpha \in (\mathbf{E} \cup \mathbf{C} \cup \mathbf{A})$ and $s \in (\mathbf{E} \cup \mathbf{C} \cup \mathbf{A})^*$ are defined as follows

- $q \Rightarrow q'$ means that there is a sequence of zero or more transitions from q to q' that are labeled by silent events, i.e., $q \xrightarrow{\langle \tau_1, \dots, \tau_n \rangle} q'$ for $\tau_1, \dots, \tau_n \in \{\tau_{alt}, \tau_{refuse}, \tau_{loop}\}$;
- $q \overset{\alpha}{\Rightarrow} q'$ means that $q \Rightarrow q_1 \xrightarrow{\alpha} q_2 \Rightarrow q'$ for some states q_1 and q_2 ;
- if $s = \langle \alpha_1, \alpha_2, \dots, \alpha_n \rangle$, then $q \overset{s}{\Rightarrow} q'$ means that $q \overset{\alpha_1}{\Rightarrow} q_1 \overset{\alpha_2}{\Rightarrow} q_2 \dots \overset{\alpha_n}{\Rightarrow} q'$.

To obtain correct action expressions for transitions, we define the function $c2g \in \mathbf{C}^* \rightarrow \mathbf{BExp}$ for converting a sequence of constraints into a guard and the function $a2ef \in \mathbf{A}^* \rightarrow (\mathbf{Var} \times \mathbf{Exp})^*$ for converting a sequence of sequence diagram assignments into a sequence of state machine assignments. More precisely, these functions are defined as follows

$$\begin{aligned}
c2g((\text{constr}(bx_1, l), \dots, \text{constr}(bx_n, l))) &\stackrel{\text{def}}{=} conj(bx_1, \dots, bx_n) \\
a2ef((\text{assign}(x_1, ex_1, l), \dots, \text{assign}(x_n, ex_n, l))) &\stackrel{\text{def}}{=} ((x_1, ex_1), \dots, (x_n, ex_n))
\end{aligned}
\tag{25}$$

where $conj \in \mathbf{BExp}^* \rightarrow \mathbf{BExp}$ yields the conjunction of a sequence of Boolean expressions. For the empty sequence, $conj$ yields true, i.e., $conj() = t$. We use the function $conj$ instead of expressing the conjunction directly because we have not defined the notation for Boolean expressions in $\mathbf{BExp}$ since this is not important in this report.

To distinguish negative from positive traces, we make use of the τ_{refuse} silent event. That is, any execution that involves a τ_{refuse} represents a negative behavior. Otherwise the execution represents positive behavior.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 27 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Definition 20 (Positive and negative experiment relations) *The relations $\Rightarrow_{pos}^s$ and $\Rightarrow_{neg}^s$ for any $s \in (\mathbf{E} \cup \mathbf{C} \cup \mathbf{A})^*$ are defined as follows*

1. $q \Rightarrow_{neg}^s q'$ means that $q \xrightarrow{t} q_1 \xrightarrow{\tau_{refuse}} q_2 \xrightarrow{u} q'$ for some states q_1 and q_2 and some traces t and u such that $s = t \frown u$
2. $q \Rightarrow_{pos}^s q'$ means that $q \xrightarrow{s} q'$ and not $q \Rightarrow_{neg}^s q'$

Since the goal of phase 1 is to construct a state machine SM that describes the negative traces of a sequence diagram, each final state of SM should accept a negative trace. To distinguish these final states from those that accept positive traces, we let each state of SM have one of two modes: *pos* and *neg*. If a state has mode *pos*, then this means that a positive trace is being recorded when this state is entered. If a state has mode *neg*, then a negative trace is being recorded when the state is entered.

Even though we shall restrict attention to well formed sequence diagrams, we cannot in general let the alphabet of the state machine be equal to the set of all events in the sequence diagram, because this set may not satisfy the requirement that all events in the alphabet must be distinct up to argument renaming (see Def. 10). To ensure that a correct alphabet is constructed, we make use of the function $\psi \in \mathbf{PVar} \rightarrow \mathbf{PVar}$ that renames parameter variables. We lift the function to signals with parameter variables as arguments as follows:

$$\psi(st(pv_1, pv_2, \dots, pv_n)) \stackrel{\text{def}}{=} st(\psi(pv_1), \psi(pv_2), \dots, \psi(pv_n))$$

To ensure that the renaming function does not change the meaning of a signal, we require that ψ does not rename two distinct parameter variables of a signal into the same parameter variable, i.e., we require

$$\forall i, j \in \{1, \dots, n\} : (\psi(st(pv_1, \dots, pv_n)) = st(pv'_1, \dots, pv'_n) \wedge pv_i \neq pv_j) \implies pv'_i \neq pv'_j \quad (26)$$

We lift ψ to expressions, events, and actions in the obvious way. Furthermore, we lift ψ to event sets and transition sets as follows:

$$\begin{aligned} \psi(E) &\stackrel{\text{def}}{=} \{\psi(e) \in \mathbf{E}_{pv} \mid e \in E\} \\ \psi(R) &\stackrel{\text{def}}{=} \{q \xrightarrow{\psi(act)} q' \mid q \xrightarrow{act} q' \in R\} \end{aligned}$$

We are now ready to define the transformation of phase 1.

Definition 21 (Phase 1) *The transformation $ph1 \in \mathbf{D} \rightarrow \mathbf{SM}$ which takes a single lifeline sequence diagram d and yields a state machine describing the negative traces of d is defined by*

$$ph1(d) = (\psi(eca.d \cap \mathbf{E}), \mathcal{Q}, \psi(\mathcal{R}), (\{d\}, pos), \{(Q, neg) \in \mathcal{Q} \mid skip \in Q\})$$

where

$$\mathcal{Q} = \mathbb{P}(\mathbf{D}) \times \{pos, neg\}$$

$\psi \in \mathbf{PVar} \rightarrow \mathbf{PVar}$ renames parameter variables such that

$$\forall e, e' \in \psi(eca.d \cap \mathbf{E}) : \neg(e = e')$$

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 28 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

and transition relation $\mathcal{R}$ is defined by the following formula

$$\begin{aligned}
\text{let } Last(d') &\stackrel{\text{def}}{=} \forall \exists ec \in (\mathbf{E} \cup \mathbf{C}) : \exists d'' \in \mathbf{D} : \Pi(ll.d, d') \xrightarrow{ec} \Pi(ll.d, d'') \\
&\quad \vee d' = \text{skip} \\
St(Q, t, mo) &\stackrel{\text{def}}{=} \{d'' \in \mathbf{D} \mid \\
&\quad d' \in Q \wedge \Pi(ll.d, d') \xrightarrow{t}_{mo} \Pi(ll.d, d'') \wedge Last(d'')\} \\
\\
\text{in } \forall tc \in \mathbf{C}^* : \forall e \in eca.d \cap \mathbf{E} : \forall ta \in \mathbf{A}^* : \\
&\quad \forall (Q, mo) \in \mathcal{Q} : \forall mo' \in \{pos, neg\} : \\
&\quad \quad \wedge St(\{d\}, ta, mo') \neq \emptyset \\
&\quad \quad \Leftrightarrow (\{d\}, pos) \xrightarrow{(\epsilon, \epsilon, ta)} (St(\{d\}, ta, mo'), mo') \in \mathcal{R} \\
&\quad \quad \wedge St(Q, tc \frown \langle e \rangle \frown ta, pos) \neq \emptyset \\
&\quad \quad \Leftrightarrow (Q, mo) \xrightarrow{(e, c2g(tc), a2ef(ta))} (St(Q, tc \frown \langle e \rangle \frown ta, pos), mo) \in \mathcal{R} \\
&\quad \quad \wedge St(Q, tc \frown \langle e \rangle \frown ta, neg) \neq \emptyset \\
&\quad \quad \Leftrightarrow (Q, mo) \xrightarrow{(e, c2g(tc), a2ef(ta))} (St(Q, tc \frown \langle e \rangle \frown ta, neg), neg) \in \mathcal{R}
\end{aligned}$$

The predicate *Last* (in Def. 21) ensures that the longest possible sequence of assignments is selected. For instance, the condition ensures that the following sequence diagram

$$\text{refuse (a seq assign}(i = 0, l) \text{ seq assign}(j = 0, l))$$

is transformed into the state machine W' in Figure 2, and not the state machine W of Figure 2. Note that the projection system consisting of those states that can be reached from the sequence diagram is illustrated at the top of Figure 2.

Each state of the state machine constructed in phase 1 consists of a set of diagrams Q rather than a single diagram which is used by the projection system. This is to reduce nondeterminism in the constructed state machine. To see how this works, consider the LTS labeled A illustrated on the left hand side of Figure 3. If we convert this into a state machine by removing silent events without merging states, we would obtain the state machine B shown in the middle of Figure 3 (note that we have omitted to specify the modes of states in the figure). Clearly, this state machine is nondeterministic. However, if we merge states 3 and 4, we obtain the state machine C (on the right hand side of Figure 3) which is deterministic.

Lemma 1 *Let d be a well formed single lifeline sequence diagram, then the state machine $ph1(d)$ describes the negative traces of d , i.e.,*

$$\llbracket ph1(d) \rrbracket = H_{neg} \quad \text{for } \llbracket d \rrbracket = (H_{pos}, H_{neg})$$

Phase 2

In phase 2, the state machine obtained from phase 1 is inverted into a state machine SM' whose semantics is the set of all traces that do not have a trace of SM as a sub-trace. This notion of inversion is captured by the following definition.

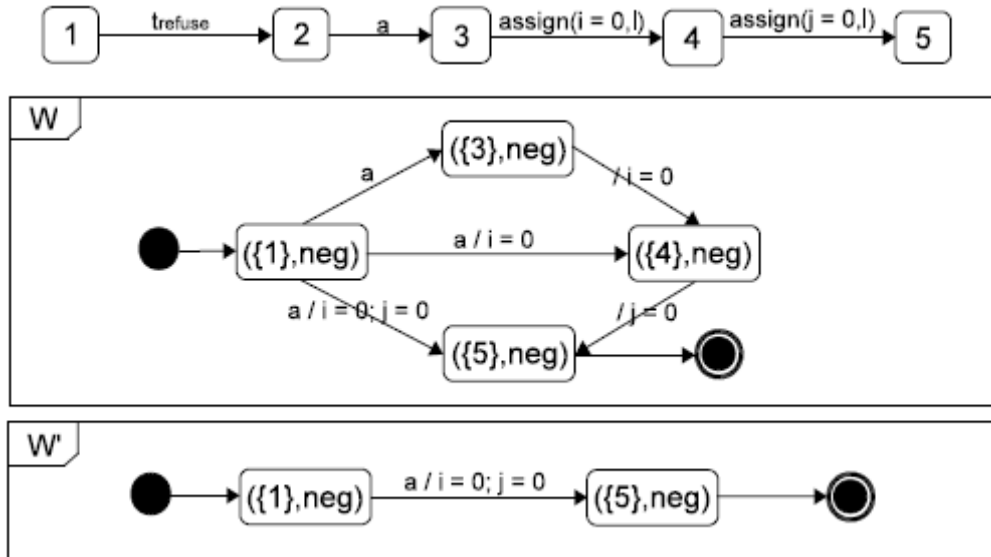


Figure 2: State machines W and W' are obtained by transformation without and with condition Last, respectively

Definition 22 (Inversion) State machine SM' is an inversion of state machine SM , written $inv(SM, SM')$, iff

$$alph(SM) = alph(SM')$$

and for all $s \in \{e \in \hat{E} \mid \exists e' \in alph(SM) : e = e'\}^*$

$$(\forall t \in \llbracket SM \rrbracket : \neg(t \triangleleft s)) \Leftrightarrow s \in \llbracket SM' \rrbracket$$

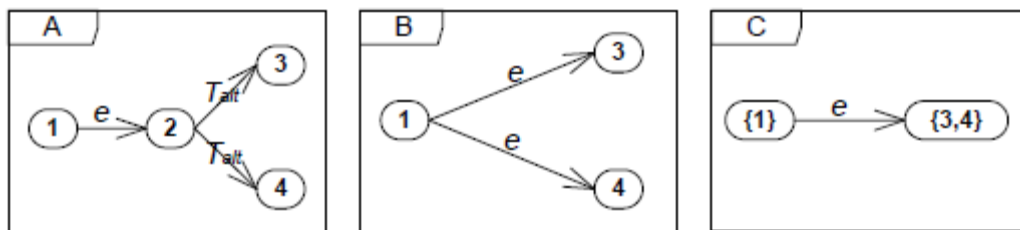


Figure 3: Machines A and B are nondeterministic while C is deterministic.

To explain how the transformation of phase 2 works, we first define the transformation for state machines whose transitions each contain exactly one event (whose signal has zero arguments) and no guards or assignments.



Concepts for Model-Based Security Testing

Deliverable ID: D2.WP2

Page : 30 of 134

Version: 1.0
Date : 26.09.2011

Status : Final
Confid : Public

Definition 23 (Phase 2 - Preliminary definition 1) *The transformation $ph2' \in \mathbf{SM} \rightarrow \mathbf{SM}$ which yields the inversion of state machines whose transitions each contain exactly one event (whose signal has zero arguments) and no guards or assignments is defined by*

$$ph2'((\mathcal{E}, \mathcal{Q}, \mathcal{R}, q_I, \mathcal{F})) \stackrel{\text{def}}{=} (\mathcal{E}, \mathbb{P}(\mathcal{Q}), \mathcal{R}', \{q_I\}, \mathbb{P}(\mathcal{Q}))$$

where the transition relation $\mathcal{R}'$ is defined by the formula

$$\begin{aligned} \text{let } St(Q, e) &\stackrel{\text{def}}{=} \{q' \in \mathcal{Q} \mid \exists q \in Q : q \xrightarrow{e} q' \in \mathcal{R}\} \\ \text{in } \forall Q \in \mathbb{P}(\mathcal{Q}) : \\ &\forall e \in \mathcal{E} : \\ &St(Q, e) \cap \mathcal{F} = \emptyset \Leftrightarrow Q \xrightarrow{e} Q \cup St(Q, e) \in \mathcal{R}' \end{aligned}$$

The rule for generating transitions ensures that previously visited states are “recorded”. To see why this is needed, consider the state machine P on the left hand side of Figure 4. The set of traces described by it is

$$\{\langle a, a \rangle, \langle b, b \rangle\}$$

The inversion of P is the state machine P' shown on the right hand side of Figure 4, i.e., $ph2'(P) = P'$. All states of P' are final, thus we have omitted to specify the final states in the figure. The trace semantics of P' is the set

$$\{\langle \rangle, \langle a \rangle, \langle b \rangle, \langle a, b \rangle, \langle b, a \rangle\}$$

Initially, both a and b are enabled. However, if a has occurred, then only b is enabled (if we assume that the alphabet of the state machine is {a, b}). Similarly if b has occurred, then only a is enabled. If both a and b have occurred, then no events are enabled. The final states of P are used to find those events that

should *not* be enabled in P'. For instance, consider the transition $\{1\} \xrightarrow{a} \{1,2\}$ in P'. Here the state 1 is “collected” because we need to make sure that b is not enabled after b has occurred in state {1,2}. Since 1 is collected, the occurrence of b in state {1,2} leads to state {1,2,3} and b is not enabled in this state because the occurrence of b in state 3 leads to a final state in P.

The following lemma shows that the transformation of phase 2 is correct for simple state machines, i.e., state machines with no guards or assignments.

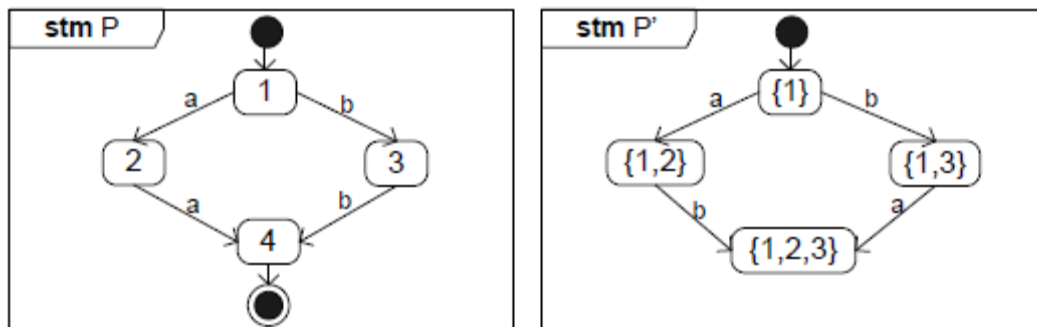


Figure 4: State machine P and its inversion P'

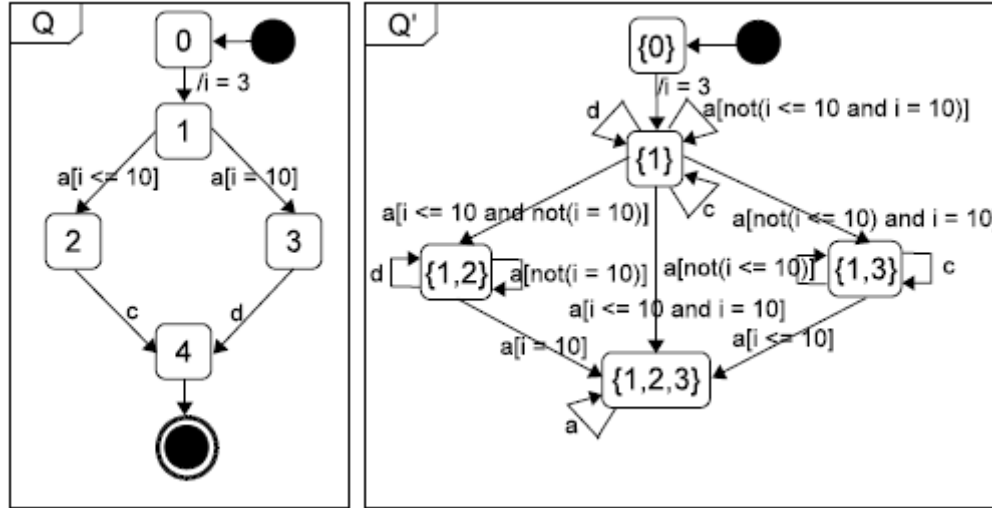


Figure 5: State machine Q and its inversion Q'

Lemma 2 *Let SM be a state machine whose transitions each contain exactly one event (whose signal has zero arguments) and no guards or assignments. Then $ph2'(SM)$ is an inversion of SM if $\langle \rangle \notin \llbracket SM \rrbracket$, i.e.,*

$$inv(SM, ph2'(SM))$$

The transformation of phase 2 is more complicated for state machines whose transitions contain guards. To see this, consider the state machine Q depicted on the left hand side of Figure 5. Inverting this state machine according to the transformation of Def. 23 would not work because the transformation does not take the guards into consideration. A correct inversion of Q is given by the state machine Q' depicted on the right hand side of Figure 5. Here we see that the transitions

$$1 \xrightarrow{(a, bx_1, \epsilon)} 2 \quad \text{and} \quad 1 \xrightarrow{(a, bx_2, \epsilon)} 3$$

of state machine Q (where bx_1 denotes $i \leq 10$ and bx_2 denotes $i = 10$) have been converted into the transitions

$$\begin{aligned} \{1\} &\xrightarrow{(a, \text{not}(bx_1 \text{ or } bx_2), \epsilon)} \{1\} & \{1\} &\xrightarrow{(a, \text{not}(bx_1) \text{ and } bx_2, \epsilon)} \{1, 3\} \\ \{1\} &\xrightarrow{(a, bx_1 \text{ and not}(bx_2), \epsilon)} \{1, 2\} & \{1\} &\xrightarrow{(a, bx_1 \text{ and } bx_2, \epsilon)} \{1, 2, 3\} \end{aligned}$$

In general, if a state machine SM has transitions

$$q \xrightarrow{(e, bx_1, sa_1)} q_1, q \xrightarrow{(e, bx_2, sa_2)} q_2, \dots, q \xrightarrow{(e, bx_n, sa_n)} q_n$$

where $q_1, \dots, q_n$ are not final states, and its inversion SM' has a state $\{q\}$, then for each set of indexes $I_x \subseteq \{1, \dots, n\}$, the inversion SM' has a transition

$$\{q\} \xrightarrow{(e, bx \text{ and } bx', sa)} \{q\} \cup Q$$

where bx is the conjunction of those guards bx_i that have an index in I_x (i.e., $i \in I_x$), bx_* is the negation of the disjunction of the guards that do *not* have an index in I_x , sa is the concatenated sequence of assignment sequences sa_i that have an index in I_x , and Q is the set of states q_i that have an index in I_x .

Note that for the special case where $I_x = \emptyset$, then bx should be equal to true. In addition, for the special case where $I_x = \{1, \dots, n\}$, then bx_* should be equal to true.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 32 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

To make this more precise, we make use of the function $- \oplus - \in \mathbb{P}(\mathbb{N}) \times A \rightarrow A$ (where $\mathbb{N}$ denotes the set of all natural numbers, and A denotes the set of all sequences), that for a set of indexes Ix and a sequence s , yields the sequence obtained from s by removing all elements whose index is not in Ix , e.g.,

$$\{1, 3, 6\} \oplus (a, b, c, d, e) = (a, c) \quad \text{and} \quad \{2, 4, 5\} \oplus (a, b, c, d, e) = (b, d, e)$$

In addition, we let *list* be a function that turns a set into a list, *set* be a function that turns a list into a set (according to some total ordering on the elements in the set), and *flatten* be the function that flattens a nested sequence, e.g.,

$$\begin{aligned} \text{list}(\{a, b, c\}) &= (a, b, c) \\ \text{list}(\{b, a, c\}) &= (a, b, c) \\ \text{set}((a, b, a, c)) &= \{a, b, c\} \\ \text{flatten}((a, (b, c), (), f)) &= (a, b, c, f) \end{aligned}$$

We also need functions on boolean expressions. As before, we let the function *conj* yield the conjunction of a sequence of boolean expressions. We also define the function $\text{disj} \in \mathbf{BExp}^* \rightarrow \mathbf{BExp}$ which yields the disjunction of a sequence of boolean expressions. For the empty sequence, *disj* yields false, i.e., $\text{disj}() = f$. Finally, we let $\text{neg} \in \mathbf{BExp} \rightarrow \mathbf{BExp}$ be the function that yields the negation of a boolean expression.

We now revise the definition of the transformation of phase 2 in light of the above discussion.

Definition 24 (Phase 2 - preliminary definition 2) *The transformation $\text{ph2}'' \in \mathbf{SM} \rightarrow \mathbf{SM}$ which yields the inversion of well formed state machines is defined by*

$$\text{ph2}''((\mathcal{E}, Q, \mathcal{R}, q_I, \mathcal{F})) \stackrel{\text{def}}{=} (\mathcal{E}, \mathbb{P}(Q), \mathcal{R}', \{q_I\}, \mathbb{P}(Q))$$

where the transition relation $\mathcal{R}'$ is defined by the following two rules:

$$q_I \xrightarrow{(\epsilon, \epsilon, sa)} q' \in \mathcal{R} \Leftrightarrow \{q_I\} \xrightarrow{(\epsilon, \epsilon, sa)} \{q'\} \in \mathcal{R}'$$

and

$$\begin{aligned} \text{let } Vi(Q, e) &\stackrel{\text{def}}{=} \{q \xrightarrow{(e', bx', as')} q' \in \mathcal{R} \mid q \in Q \wedge e = e'\} \\ Vi(Q, e, Ix) &\stackrel{\text{def}}{=} \text{set}(Ix \oplus \text{list}(Vi(Q, e))) \\ St(Q, e, Ix) &\stackrel{\text{def}}{=} \{q \in Q \mid \exists q' \xrightarrow{(e', bx', as')} q'' \in Vi(Q, e, Ix) : q = q''\} \\ Ga(Q, e, Ix) &\stackrel{\text{def}}{=} \text{list}(\{bx \in \mathbf{BExp} \mid \\ &\quad \exists q' \xrightarrow{(e', bx', as')} q'' \in Vi(Q, e, Ix) : bx = bx'\}) \\ Ga'(Q, e, Ix) &\stackrel{\text{def}}{=} \text{conj}((\text{conj}(Ga(Q, e, Ix), \\ &\quad \text{neg}(\text{disj}(Ga(Q, e, \mathbb{N} \setminus Ix)))))) \\ As(Q, e, Ix) &\stackrel{\text{def}}{=} \{as \in (\mathbf{Var} \times \mathbf{Exp})^* \mid \\ &\quad \exists q' \xrightarrow{(e', bx', as')} q'' \in Vi(Q, e, Ix) : as = as'\} \\ As'(Q, e, Ix) &\stackrel{\text{def}}{=} \text{flatten}(\text{list}(As(Q, e, Ix))) \\ \text{in } \forall Q \in \mathbb{P}(Q) : \\ &\quad \forall e \in \mathcal{E} : \\ &\quad \quad \forall Ix \in \mathbb{P}(\mathbb{N}) : \\ &\quad \quad \quad St(Q, e, Ix) \cap \mathcal{F} = \emptyset \Leftrightarrow \\ &\quad \quad \quad Q \xrightarrow{(e, Ga'(Q, e, Ix), As'(Q, e, Ix))} (Q \cup St(Q, e, Ix)) \in \mathcal{R}' \end{aligned}$$

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 33 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

The transformation $ph2''$ does not always yield the correct inversion of a state machine. For instance, consider the state machine W depicted on the left hand side of Figure 6. It describes two traces: one trace consisting of 9 occurrences of a , and one trace consisting of 9 occurrences of b . Applying the phase 2 transformation $ph2''$ to W yields the state machine W' depicted on the right hand side of Figure 6. Note that we have not depicted final states (since all states are final) or transitions whose guards always evaluate to false and that redundancy in the boolean expressions of the guards have been removed, e.g., $i \leq 10$ and true is written $i \leq 10$.

The state machine W' rejects traces consisting of 10 or more occurrences of a or b . For instance, the trace t consisting of 5 occurrences of a and 5 occurrences of b is rejected by the state machine W' . However, no trace of W is a sub-trace of t . Hence, W' is not a correct inversion of W . The reason for this is that the two possible executions of W , resulting from the branch in state 2, *share* the variable i , i.e., the variable is used in a condition/guard of one execution and assigned to a value in another execution. In the example, this causes W' not to be a correct inversion of W .

In general, to ensure that $ph2''$ yields the correction inversion $SM_{\perp}$ of a state machine SM , we must require that all guards encountered when executing $SM_{\perp}$ must evaluate to the same values as the "corresponding" guards encountered when executing SM . We say that a transformation is *side effect free* for SM if this requirement is satisfied. This is precisely captured by the following definition.

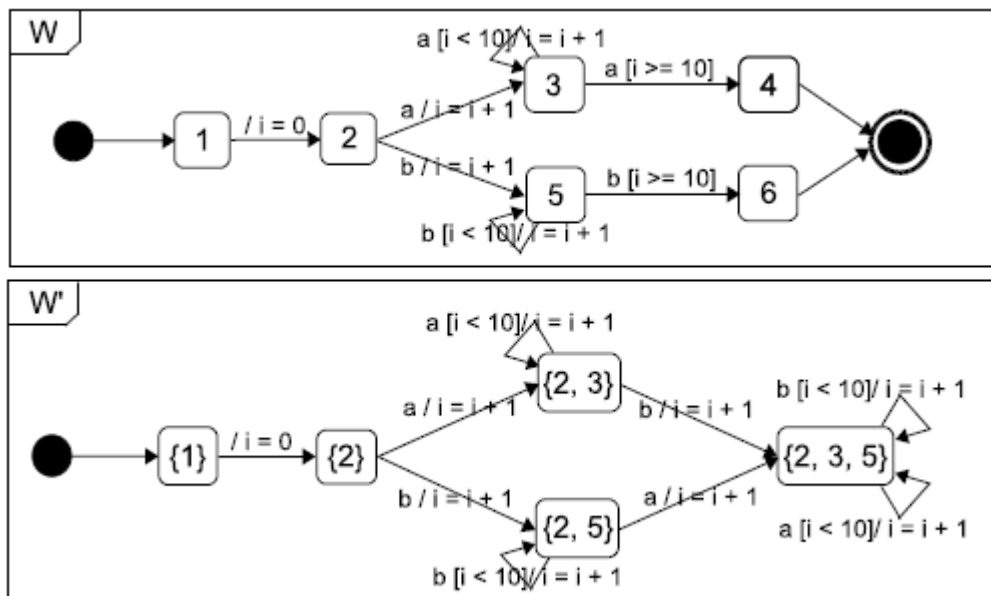


Figure 6: State machine W and its inversion

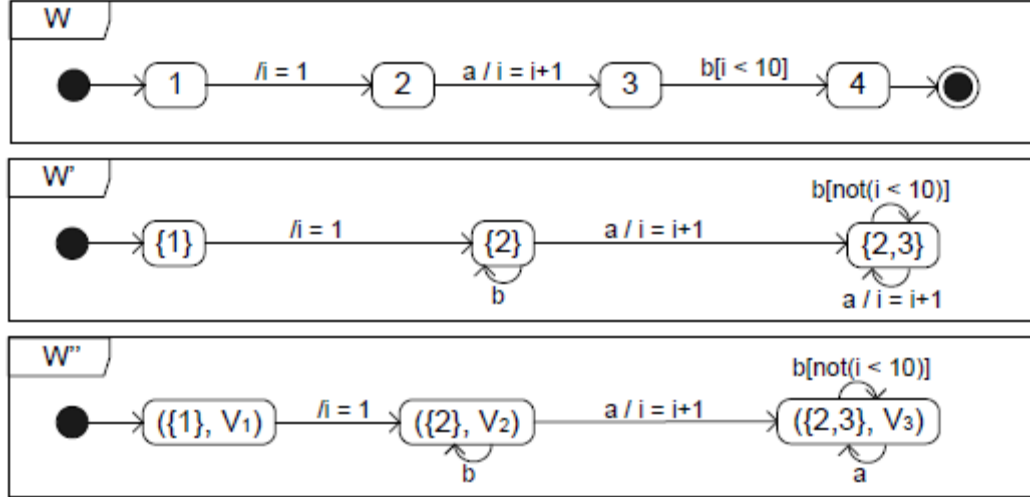


Figure 7: State machine W, its incorrect inversion $ph2(W) = W'$, and its correct inversion W''

Definition 25 (Side effect free) Let $SM = (\mathcal{E}, Q, \mathcal{R}, q_I, \mathcal{F})$, $tr \in \mathbf{SM} \rightarrow \mathbf{SM}$, and $tr(SM) = SM' = (\mathcal{E}', Q', \mathcal{R}', q'_I, \mathcal{F}')$. Then transformation tr is side effect free for SM iff

$$\begin{aligned}
 & \forall s, t \in (\hat{\mathbf{E}} \cup \{\epsilon\})^* : \forall (e, bx, as) \in \mathbf{Act}_{\hat{\mathbf{E}}} : \\
 & \quad \forall q_1, q_2 \in Q : \forall q'_1 \in Q' : \\
 & \quad \quad \forall \sigma_I, \sigma_1, \sigma'_1, \sigma'_1 \in \hat{\Sigma}_T : \\
 & \quad \quad \quad \wedge [q_I, \sigma_I] \xrightarrow{s} [q_1, \sigma_1] \in EG(SM) \\
 & \quad \quad \quad \wedge [q'_I, \sigma'_I] \xrightarrow{t} [q'_1, \sigma'_1] \in EG(SM') \\
 & \quad \quad \quad \wedge q_1 \xrightarrow{(e, bx, as)} q_2 \in \mathcal{R} \\
 & \quad \quad \quad \wedge comp(\llbracket SM \rrbracket, s, t) \\
 & \quad \quad \quad \implies \sigma_1 \cap (var(bx) \times \mathbf{Exp}) = \sigma'_1 \cap (var(bx) \times \mathbf{Exp})
 \end{aligned}$$

where the predicate $comp(_, _, _) \in \mathbb{P}(\mathbf{E}^*) \times \mathbf{E}^* \times \mathbf{E}^* \rightarrow \mathbb{B}$ is defined $comp(T, s, t) \stackrel{\text{def}}{=} s \triangleleft t \wedge pr(T, s) \neq \emptyset \wedge \neg(\exists s' \in pr(T, s) : s \sqsubset s' \wedge s' \triangleleft t)$ where the function $pr \in \mathbb{P}(\mathbf{E}^*) \times \mathbf{E}^* \rightarrow \mathbb{P}(\mathbf{E}^*)$ is defined by $pr(T, s) \stackrel{\text{def}}{=} \{s \sim s' \in T \mid \neg(\exists t \in T : t \sqsubset s \sim s')\}$.

Lemma 3 Let SM be a well formed state machine such that $\langle \rangle \notin \llbracket SM \rrbracket$ and $ph2''$ be side effect free for SM , then $ph2''(SM)$ is an inversion of SM , i.e.,

$$inv(SM, ph2''(SM))$$

It is possible to define an alternative version of the transformation of phase 2 for which the side effect free condition is less restrictive. In particular, we observe that $ph2''$ may generate unnecessary guards and assignment sequences for transitions corresponding to inconclusive behavior. As an example, consider the state machine W in Figure 7. It describes the trace $\langle a, b \rangle$. The result of applying transformation $ph2''$ to W is depicted by state machine W' in Fig.17 (i.e., $ph2''(W) = W'$). Note that we have not illustrated final states (since all states are final) or transitions whose guards always evaluate to false, and that boolean expressions



Concepts for Model-Based Security Testing

Deliverable ID: D2.WP2

Page : 35 of 134

Version: 1.0

Date : 26.09.2011

Status : Final

Confid : Public

have been simplified. The state machine W' is not a correct inversion of W since b is enabled after a has occurred 10 times. The problem is that the reflexive transition

$$\{2, 3\} \xrightarrow{(a/i = i + 1)} \{2, 3\}$$

in W' describing inconclusive behavior, contains the (unnecessary) assignment $i = i + 1$ since W has the transition

$$2 \xrightarrow{(a/i = i + 1)} 3$$

which has been previously visited to reach the state $\{2, 3\}$.

A solution to the problem of the current example, is to let each state of the inverted state machine record all transitions that are previously visited in order to reach that state. The previously visited transitions can then be disregarded when generating reflexive transitions corresponding to inconclusive behavior.

State machine W'' in Figure 7 shows how this would work in the current example. Here V_1, V_2, V_3 are sets of previously visited transitions of W defined by

$$V_1 = \emptyset \quad V_2 = \{1 \xrightarrow{/i = 1} 2\} \quad V_3 = \{1 \xrightarrow{/i = 1} 2, 2 \xrightarrow{a/i = i + 1} 3\}$$

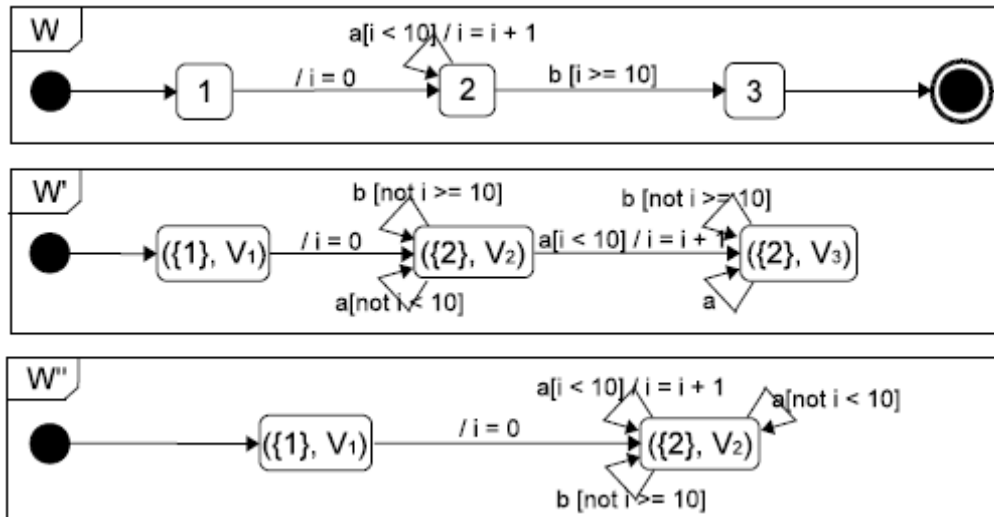


Figure 8: State machine W and its (incorrect) inversion W' and (correct) inversion W''

Now, when generating transitions for a in state $(\{2, 3\}, V_3)$, we disregard the set V_3 of previously visited transitions. Thus we get

$$(\{2, 3\}, V_3) \xrightarrow{a} (\{2, 3\}, V_3)$$

and by definition of inversion (Def. 22) we have that W'' is a correct inversion of W .

The solution proposed above may not work for state machines that contain loops. For instance, consider the state machine W of Figure 8. It describes the trace containing 9 occurrences of a followed by b . In other words, the policy states that b is not allowed to occur after a has occurred 9 times. If we use the transformation $ph2''$ to invert W and record previously visited transitions as described above, we get state machine W' of Figure 8. Here we have that

$$V_1 = \emptyset \quad V_2 = \{1 \xrightarrow{/i=0} 2\} \quad V_3 = \{1 \xrightarrow{/i=0} 2, 2 \xrightarrow{a[i < 10]/i=i+1} 2\}$$

The state machine W' is not a correct inversion of W since it allows the occurrence of b after a has occurred

more than 9 times. In this case, adding the transition $2 \xrightarrow{(a, i < 10, i = i + 1)} 2$ into the set of previously visited transitions, and thereby disregarding its transitions, is incorrect, because the transition is in a loop and may therefore be visited several times.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 36 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

A solution to the problem is to remove the transitions of a loop from the set of visited transitions each time the loop is iterated. To achieve this, we can remove the outgoing transitions of each state that is entered from the set of previously visited transitions. In the current example, we would then obtain the state machine W'' of Figure 8 which is a correct inversion of W .

We are now ready to give the final definition of the transformation of phase 2.

Definition 26 (Phase 2) *The transformation $ph2 \in \mathbf{SM} \rightarrow \mathbf{SM}$ which yields the inversion of well formed state machines is defined by*

$$ph2((\mathcal{E}, Q, \mathcal{R}, q_I, \mathcal{F})) \stackrel{\text{def}}{=} (\mathcal{E}, (\mathbb{P}(Q) \times \mathbb{P}(\mathcal{R})), \mathcal{R}', (\{q_I\}, \emptyset), (\mathbb{P}(Q) \times \mathbb{P}(\mathcal{R})))$$

where the transition relation $\mathcal{R}'$ is defined by the following two rules:

$$q_I \xrightarrow{(\epsilon, \epsilon, sa)} q' \in \mathcal{R} \Leftrightarrow (\{q_I\}, \emptyset) \xrightarrow{(\epsilon, \epsilon, sa)} (\{q'\}, \emptyset) \in \mathcal{R}'$$

and

$$\begin{aligned}
\text{let } Vi(Q) & \stackrel{\text{def}}{=} \{q \xrightarrow{(e', bx', as')} q' \in \mathcal{R} \mid q \in Q\} \\
Vi(Q, e, V) & \stackrel{\text{def}}{=} \{q \xrightarrow{(e', bx', as')} q' \in \mathcal{R} \mid q \in Q \wedge e = e'\} \setminus V \\
Vi(Q, e, Ix, V) & \stackrel{\text{def}}{=} set(Ix \oplus list(Vi(Q, e, V))) \\
St(Q, e, Ix, V) & \stackrel{\text{def}}{=} \{q \in Q \mid \\
& \quad \exists q' \xrightarrow{(e', bx', as')} q'' \in Vi(Q, e, Ix, V) : q = q''\} \\
Ga(Q, e, Ix, V) & \stackrel{\text{def}}{=} list(\{bx \in \mathbf{BExp} \mid \\
& \quad \exists q' \xrightarrow{(e', bx', as')} q'' \in Vi(Q, e, Ix, V) : bx = bx'\}) \\
Ga'(Q, e, Ix, V) & \stackrel{\text{def}}{=} conj((conj(Ga(Q, e, Ix, V), \\
& \quad neg(disj(Ga(Q, e, \mathbb{N} \setminus Ix, V)))))) \\
As(Q, e, Ix, V) & \stackrel{\text{def}}{=} \{as \in (\mathbf{Var} \times \mathbf{Exp})^* \mid \\
& \quad \exists q' \xrightarrow{(e', bx', as')} q'' \in Vi(Q, e, Ix, V) : as = as'\} \\
As'(Q, e, Ix, V) & \stackrel{\text{def}}{=} flatten(list(As(Q, e, Ix, V))) \\
\text{in } \forall (Q, V) \in \mathbb{P}(Q) \times \mathbb{P}(\mathcal{R}) : \forall e \in \mathcal{E} : \\
& \quad \forall Ix \in \mathbb{P}(\mathbb{N}) : \\
& \quad St(Q, e, Ix, V) \cap \mathcal{F} = \emptyset \Leftrightarrow \\
& \quad (Q, V) \xrightarrow{(e, Ga'(Q, e, Ix, V), As'(Q, e, Ix, V))}, \\
& \quad (Q \cup St(Q, e, Ix, V), (V \cup Vi(Q, e, Ix, V)) \setminus Vi(St(Q, e, Ix, V))) \in \mathcal{R}'
\end{aligned}$$

Corollary 1 *Let SM be a well formed state machine such that $\langle \rangle \notin \llbracket SM \rrbracket$ and $ph2$ be side effect free for SM , then $ph2(SM)$ is an inversion of SM , i.e.,*

$$inv(SM, ph2(SM))$$

The transformation $ph2$ will correctly invert the state machine examples of Figure 4, Figure 5, Figure 7, and Figure 8.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 37 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

However, *ph2* does not work for the state machine of Figure 6, where the variable *i* is *shared* in the sense that it is used in a condition/guard of one execution and assigned to a value in another execution. We make this precise in the following definition.

Definition 27 (Shared variables) *Let $SM = (\mathcal{E}, \mathcal{Q}, \mathcal{R}, q_I, \mathcal{F})$, then SM does not have any shared variables iff*

$$\begin{aligned}
& \forall s, t, t' \in \mathbf{Act}^* : \forall q, q' \in \mathcal{Q} : \\
& \quad \forall (e_1, bx_1, as_1), (e'_1, bx'_1, as'_1), (e_2, bx_2, as_2), (e'_2, bx'_2, as'_2) \in \mathbf{Act} : \\
& \quad \wedge q_I \xrightarrow{s \neg \langle (e_1, bx_1, as_1) \rangle \neg t \neg \langle (e_2, bx_2, as_2) \rangle} q \in \mathcal{R} \\
& \quad \wedge q_I \xrightarrow{s \neg \langle (e'_1, bx'_1, as'_1) \rangle \neg t' \neg \langle (e'_2, bx'_2, as'_2) \rangle} q' \in \mathcal{R} \\
& \quad \wedge (e_1, bx_1, as_1) \neq (e'_1, bx'_1, as'_1) \\
& \quad \wedge \neg (\forall \sigma \in \hat{\Sigma}_T : eval(\sigma(bx_1)) = eval(\sigma(bx'_1))) \\
& \quad \implies ((var(bx_2) \cap avar(as'_2)) \setminus \mathbf{PVar}) = \emptyset
\end{aligned}$$

where $avar \in (\mathbf{Var} \times \mathbf{Exp})^* \rightarrow \mathbb{P}(\mathbf{Var})$ yields all the variables that are assigned to a value in an assignment sequence, i.e., $avar(((x_1, ex_n), \dots, (x_n, ex_n))) = \{x_1\} \cup \dots \cup \{x_n\}$.

We conjecture that if a state machine SM does not have shared variables in the sense of (Def. 27), then *ph2* is side effect free for SM (Def. 25). By Corollary 1, this means *ph2* will yield the correct inversion of any state machine that does not have shared variables.

In practice, the condition that a state machine policy must not have shared variables, means that when we compose several policies, then these policies cannot have the same variable names. For instance, consider again the state machine W of Figure 6. This state machine may be seen as the composition of the two policies: (1) more than 9 occurrences of *a* is not allowed, and (2) more than 9 occurrences of (2) *b* is not allowed. However, since both policies use the variable *i* to count the number of occurrences of *a* or *b*, the condition of no shared variables is violated.

Note that it is feasible to automatically check whether a state machine has no shared variables because the condition is formulated in terms of the transitions of a state machine as opposed to the transitions of the execution graph.

Together, the transformation of phase 1 and phase 2 is adherence preserving when the condition of phase 2 is satisfied.

Theorem 1 *Let d be a well formed single lifeline sequence diagram such that *ph2* is side effect free for $ph1(d)$, then the transformation $d2p(d)$ is adherence preserving, i.e.,*

$$d \rightarrow_{da} \Phi \Leftrightarrow d2p(d) \rightarrow_{sa} \Phi \quad \text{for all systems } \Phi$$

1.2.4.2 From general sequence diagrams to state machines

In this section, we define the transformation that takes a (general) sequence diagram and yields a set of state machines.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 38 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Definition 28 (From sequence diagrams to sets of state machines) *The transformation $d2pc \in \mathbf{D} \rightarrow \mathbb{P}(\mathbf{SM})$ which takes a sequence diagram and yields a set of state machine, is defined by*

$$d2pc(d) \stackrel{\text{def}}{=} \bigcup_{l \in ll.d} \{d2p(\pi_l(d))\}$$

The transformation from sequence diagrams to state machine sets is adherence preserving when the condition of phase 2 is satisfied.

Theorem 2 *Let d be a well formed sequence diagram such that $ph2$ is side effect free for $ph1(\pi_l(d))$ for all lifelines l in d , then the transformation $d2pc(d)$ is adherence preserving, i.e.,*

$$d \rightarrow_{dag} \Phi \Leftrightarrow d2pc(d) \rightarrow_{sag} \Phi \quad \text{for all systems } \Phi$$

1.3 NETWORK INSTRUMENTATION AND DATA EXTRACTION

To understand how a system works, there must be a way to collect data from it. When discussing information networks, the venerable OSI model remains as the most commonly used way of describing network architectures. In the OSI model, network protocols are split into seven layers often denoted L1-L7, with each layer communicating only with the layers below and above it. From an implementation and traffic analysis point of view this is completely inadequate, but it serves as a starting point for discussion.

Data collection can be approached from both ends of the layered stack. The applications on Layer 7 (L7) are ultimately what define what is transmitted on the network. The applications typically interface with the network through well-defined operating system services. The applications then serialize memory representations of the necessary objects, and pass them on to the operating system for transmission. The receiver side application receives the serialized representation, and transforms it to its own internal memory representation. If both applications share an identical understanding on the data, the communication has been successful.

The downside of application level monitoring is that implementing instrumentation has to be done separately for each application, and thus requires a significant amount of resources. Also, there may be problems on the lower levels that are impossible to diagnose in the application, which sees only symptoms of the underlying problem.

If the bottom-up approach is taken, and data is collected at the lowest level, it obviously contains all the information that is necessary to understand what is being transmitted. Barring storage costs, the monetary cost to collect the data is very low. Interpreting the data, however, requires more effort. In some cases it may even be necessary to reproduce a complete implementation of the entire application stack to be able to understand the data. An example of this is application level encryption. Without reassembling and decrypting the encrypted packets analysis is limited to finding that some communication occurred between two hosts, and some correlation can be found in relation to the amount of data transferred, as well as timing. Decryption also requires infrastructure for retrieving and inserting the necessary keys.

1.3.1 Physical Level (L1)

At the lowest level, the bit stream of data that is transferred through the network is encoded into the physical realm: electrical impulses, light, radio signals etc. The range of network technologies is wide and varied, making detailed analysis of what can be collected outside the scope of this work. Furthermore, as the physical phenomena must be converted into digital form for analysis, data collection is typically done at Layer 2 (or higher), where this has already been done.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 39 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

There are several useful attributes that can only be measured from the physical level, such as bit error rate and signal strength. This information is often made available by L2 analyzers, which mitigates the need for L1 analysis.

There are a few areas, where data collection at the physical level is warranted. The most natural area is in low-level conformance testing of network equipment. In wireless networks, there are also interesting characteristics that can be inferred from the physical level. Even though equipment from different vendors all implements the same standards, there are measurable differences in the RF spectrum they produce. This can be used for device fingerprinting, which has some value in, e.g., information security purposes.

1.3.2 Data Link (L2)

The most straightforward and prevalent source of data in a network are packet captures. Collecting data at this level also has the lowest monetary cost, due to non-intrusiveness and implementation simplicity. It requires the traffic to be monitored to be available to the monitoring host, which can be done in a passive manner using, e.g., monitor port functionality available in nearly all Ethernet switches, or commercially available passive taps.

Packet capturing differs from normal networking in one crucial way. It may involve traffic that it neither originating nor destined to the capturing host. Typically such packets are dropped by the network interface card (NIC), which have a small (32 or 64 entries is typical) programmable address filter. Packets that are not destined to an address matched by the filters are dropped, making host processing unnecessary. In order for all packets to be captured, the NIC can be placed into promiscuous mode, which disables the filter entirely.

Instead, filtering is often done by small virtual machine –based application, such as the Berkeley Packet Filter (BPF) used by the widely used libpcap library. These applications are loaded into kernel space by the monitoring application, and each packet is run through the registered filters. Only matching packets are then copied to the monitoring application for further processing.

The downside to packet level analysis is understanding the data. Even a simple operation, such as searching for a string from the traffic, can be overly complex, as the string may have been divided between two separate packets. Tools for reconstructing the flows, such as Wireshark and tcpflow, are widely available, but still they are limited in what they can accomplish, as fully understanding the data ultimately requires a full implementation of the overlying protocol stack.

1.3.3 Transport Layer (L3)

An alternative for collecting application traffic is from the transport layer, which provides for a reconstructed view to the data flows generated by networked applications, typically a TCP or UDP connection between two given IP address - port combinations. There are two main approaches, transport layer proxies and flow-level analysis.

1.3.3.1 Transport layer proxies

While the traffic flows can be reconstructed from the original packets on the wire, this can be a laborious task. When reconstructing, e.g., TCP connections, the packets can arrive out-of-order, be fragmented or retransmitted, or even be lost, as they arrive at the monitoring station. Reconstruction fundamentally requires a full, stateful implementation of the original protocol, which ideally covers all corner-cases in the protocol specification.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 40 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Alternatively, the monitoring point can operate as a man in the middle that terminates the original connection, and originates a new connection to the original destination host (transparent proxy). This can be advantageous, as it simplifies collection, and also makes it possible to modify the data as it passes through, if required for robustness testing of networked software. The major downside of this approach is that it is not passive, and may change the semantics of the original traffic.

1.3.3.2 Flow-based traffic collection

An alternative to capturing packet payload is only capturing summarized information of the traffic. The dominant method for this is NetFlow, originally defined by Cisco. NetFlow was designed for capturing summarized network data from routers, taking into account the limited resources available on a router.

In NetFlow, traffic is represented as an unidirectional sequence of packets with an identical 7-tuple (source and destination IP address, source and destination ports and IP protocol, Ingress physical port and IP Type of Services). As routers classify packets based on these attributes as a normal part of their functionality, implementing NetFlow is a matter of maintaining a small flow cache that is updated for each packet incrementing packet and data transmitted counts. To conserve limited resources, this cache is regularly flushed to an external collector.

There are several different versions of the NetFlow defined, with varying levels of detail provided in the record format. In practice, two versions are currently in wide use, versions 5 and 9. The most recent version of NetFlow, version 9, is an extensible, template-based format, which currently defines 89 field types (e.g. MPLS labels, IPv6 addresses and AS numbers associated with the data). The router can be configured to emit NetFlow records with a specific set of fields, as specified by the user.

While the detail available in NetFlow is limited, it has the benefit of being very compact, even for large enterprise networks storing months of data is feasible. The detail level is enough for some intrusion detection type of analysis, and can be used for post-mortem analysis of intrusions even months after the intrusion has happened.

1.4 ADVANCED NETWORK TRAFFIC ANALYSIS

Unlike active monitoring [1][58], passive monitoring does not inject traffic in the network or modify the traffic that is being transmitted in the network. This is crucial because any injected message/packet may modify the environment, causing problems or crashes of the tested protocol or service. Thus, the passive monitoring approach seems to be the ideal means for directly verifying an implementation in natural operational run-time conditions. In addition, with this approach, the tests can be run during the whole protocol life time as opposed to active testing campaigns that are run during a limited period of time.

This kind of passive monitoring can be very relevant in many fields and for different purposes (e.g. functional, security, performance). In this section, we present some security monitoring concepts for advanced network traffic monitoring based on “MMT-Security properties”¹.

1.4.1 MMT-Security Properties

1.4.1.1 Basics

¹ MMT-Security properties are used as input to Montimage Monitoring Tool (MMT) described in WP3.D2 section 5.1.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 41 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

The main objective of MMT-Security properties is to formally specify security goals and attack behaviors related to the application or protocol under test. The “MMT-Security property” model is inspired from LTL logic [23] and can refer to two types of properties: “Security rules” and “Attacks” described as follows:

- A Security rule describes the expected behavior of the application or protocol under-test whether it is functional or security oriented. The non-respect of the MMT-Security property indicates an abnormal behavior, e.g. the access to a specific service must always be preceded by an authentication phase.
- An Attack describes a malicious behavior whether it is an attack model, a vulnerability or a misbehavior. Here, the respect of the MMT-Security property indicates the detection of an abnormal behavior that might indicate the occurrence of an attack, e.g. a big number of requests from the same user in a limited period of time can be considered as a behavioral attack.

Notice that the events that we take into account within MMT-Security properties are related to observable system/network communications. In the case of a telecommunication network, they refer to traffic packets and flows. In other contexts, they can relate to any action that can be stored in a server/database/software log file. In the following, we formally present the concepts of MMT-Security properties in the context of telecommunication networks [48]. The main definition of an MMT-Security property is provided by definition number 11. The other definitions allow understanding the basics of the used model.

Definition 1 (Trace): A collected trace during a period of time is a set of ordered captured packets.

- A trace $T = \bigcup_{i=0}^n p_i$ where n is the number of the captured packets, p_1 is the first packet captured in the trace and p_n is the last one.
- Each packet p_i has a rank r_i that corresponds to its position in the trace T .
- $\forall p_i \in T, p_i = \bigcup_{j=1}^{m_i} f_{i,j}$ where $f_{i,j}$ is a field of the packet p_i and m_i is the number of fields of the packet p_i . Each field $f_{i,j}$ of the packet p_i has a value $v_{i,j}$.
- $\forall p_i \in T, \exists f_{i,j} \in p_i / f_{i,j} = t_i$ where t_i is the timestamp when p_i was captured.
- $\forall r_i, r_j$ where r_i is rank of p_i and r_j is rank of p_j , if $r_i > r_j$ then $t_i > t_j$.

Definition 2 (Value function Φ): Let T be a collected trace of n packets, F the set of fields of all the packets p_i of the trace T and V the domain of values. $V = R \cup S \cup \text{NULL}$ where S is a finite set of strings and R is the real domain values. We define the function: $\Phi: T \times F \rightarrow V$ as the function that allows providing the value of a field in a specific packet of the trace T :

- $\Phi(p_i, f_{m,n}) = v_{i,n}$ if $f_{m,n} \in p_i$ and
- $\Phi(p_i, f_{m,n}) = \text{NULL}$ if $f_{m,n} \notin p_i$

An MMT-Security property is an IF-THEN property. It allows expressing specific constraints on network events. Each event is a set of conditions on some of the field values of the exchanged packets.

Definition 3 (Conditions): Conditions are predicates on packet fields' values. Let p_i and $p_{i'}$ be two captured packets, V be the domain of values, $f_{i,j}$ be a field of the packet p_i , $f_{i',j'}$ be a field of $p_{i'}$ and $x \in V$. Let “op” be an operator element of $O_R \cup O_S$ where $O_R = \{\leq, \geq, =, \neq, \in, \text{etc.}\}$ and $O_S = \{\text{equal, not equal, contain, not contain, etc.}\}$.²

² O_R is the classical set of operators that can be applied on real numbers in the domain R .
 O_S is the classical set of operators that can be applied on strings of the domain S .

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 42 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Two types of conditions can be defined: cs (simple condition) and cc (complex condition).

- $cs ::= \Phi(p_i, f_{i,j}) \text{ op } x$

We say that the packet p_i satisfies the simple condition “cs” if and only if the predicate “ $v_{i,j} \text{ op } x$ ” is true.

- $cc ::= \Phi(p_i, f_{i,j}) \text{ op } \Phi(p_{i'}, f_{i',j'})$

We say that packet p_i satisfies the complex condition “cc” if and only if the predicate “ $v_{i,j} \text{ op } v_{i',j'}$ ” is true.

Definition 4 (Basic event): An event e_i is a set of conditions on relevant fields of captured packets.

$$e_i = \bigcup_{k=1}^{m_j} c_{i,k} \text{ where } m_j \text{ is the number of conditions (simple and/or complex).}$$

Let p_i be a packet and e_i an event with m_i conditions and $c_{i,k}$ the k^{th} condition of e_i . A packet p_i satisfies an event e_i if and only if $\forall k \in [1, m_i]$, $c_{i,k}$ is true.

Definition 5 (Abstention of having an event): If e is an event, then $\neg e$ is also an event. $\neg e$ is satisfied if no packet that satisfies the event e occurs in the collected trace.

Definition 6 (Repetition of an event): If e is an event and $n \in \mathbb{N}^+$, then $n \times e$ is a complex event. $n \times e$ is satisfied if n packets satisfying the event e occur in the collected trace.

Definition 7 (Complex events: Successive events): Let $n \in \mathbb{N}^+$, $t \in \mathbb{R}^{+*}$ and e_1 and e_2 be two basic events. $(e_1; e_2)_{n,t}$ is a complex event. $[p_1, p_2]$ satisfies $(e_1; e_2)_{n,t} \Leftrightarrow$

- p_1 satisfies e_1 and
- p_2 satisfies e_2 and
- $\text{time}(p_1) < \text{time}(p_2) < \text{time}(p_1) + t$ and
- $\text{rank}(p_1) < \text{rank}(p_2) < \text{rank}(p_1) + n$

In other words, $[p_1, p_2]$ satisfies $(e_1; e_2)_{n,t}$ iff p_2 follows p_1 and they are separated by at most n packets and t units of time.

Definition 8 (Complex events: AND): Let $n \in \mathbb{N}^+$, $t \in \mathbb{R}^{+*}$ and e_1 and e_2 two basic events. $(e_1 \wedge e_2)_{n,t}$ is a complex event. $[p_1, p_2]$ satisfies $(e_1 \wedge e_2)_{n,t}$ if $[p_1, p_2]$ satisfies $(e_1; e_2)_{n,t}$ or $(e_2; e_1)_{n,t}$.

Intuitively, p_1 and p_2 satisfy $(e_1 \wedge e_2)_{n,t}$ iff p_2 and p_1 are separated by at most n packets and t units of time.

Definition 9 (complex events: OR): Let $n \in \mathbb{N}^+$, $t \in \mathbb{R}^{+*}$ and e_1 and e_2 two basic events. $(e_1 \vee e_2)_{n,t}$ is a complex event. p_1 satisfies $(e_1 \vee e_2)_{n,t} \Leftrightarrow p_1$ satisfies e_1 or p_1 satisfies e_2 .

Definition 10 (MMT-Security property): Let $W \in \{\text{BEFORE}, \text{AFTER}\}$, $n \in \mathbb{N}^+$, $t \in \mathbb{R}^{+*}$ and e_1 and e_2 two events (basic or not). An MMT-security property is an IF-THEN expression that describes constraints on network events captured in a trace $T = \{p_1, \dots, p_m\}$. It has the following syntax:

$$e_1 \xrightarrow{W, n, t} e_2$$

This property expresses that if the event e_1 is satisfied (by one or several packets p_i , $i \in \{1, \dots, m\}$), then event e_2 must be satisfied (by a set of packets p_i , $j \in \{1, \dots, m\}$) before or after (depending on the W value) at most n packets and t units of time. e_1 is called triggering context and e_2 is called clause verdict.

The MMT-Security property model allows expressing complex security properties derived from security best practises and from domain-specific security requirements. These MMT-Security properties are described using an XML format to make interpretation easier for both humans and software. The XML structure of MMT-Security properties is presented in WP3.D2 document in section 5.1 [78]. The processing of MMT-

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 43 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Security properties is performed by the MMT tool that allows the analysis of collected communication traces in order to detect potential vulnerabilities and security flaws.

1.4.1.2 Examples

1.4.1.2.1 ARP poisoning

Address Resolution Protocol (ARP) is a telecommunications protocol used for resolution of network layer addresses into link layer addresses, a critical function in multiple-access networks. ARP was defined by RFC 826 in 1982 [12]. ARP poisoning (RFC5227) is a technique used to attack a local-area network (LAN). ARP poisoning may allow an attacker to intercept data frames on a LAN, modify the traffic, or stop the traffic altogether.

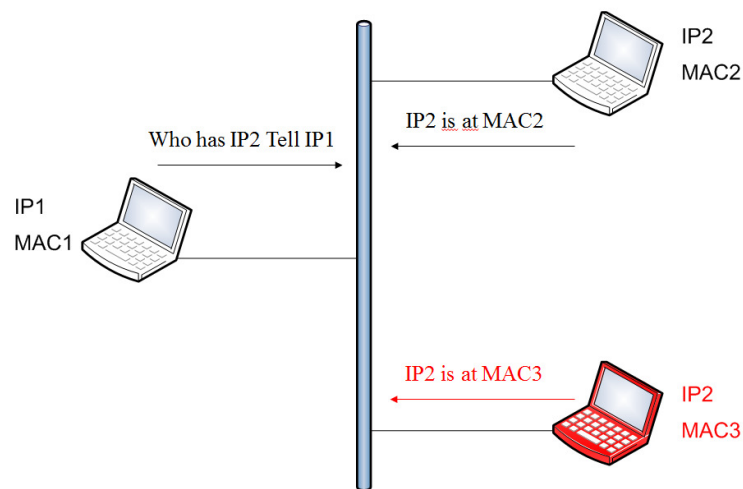


Figure 9: IP duplication

Attack model: Following an ARP request (who has), a node receives more than one reply with different MAC addresses then IP duplication is detected or potentially an ARP poisoning attack.

This attack behavior can be specified as follows:

$$(e_1 ; e_2)_{-1,1} \xrightarrow{\text{AFTER}, -1,1} e_3$$

Where:

- Event 1 (e_1): reception of ARP request message
- Event 2 (e_2): reception of ARP reply message (src mac2)
- Event 3 (e_3): reception of ARP reply message with (src mac3) $\neq$ (src mac2)

This security attack is specified in XML³ as follows:

```
<property value="THEN" delay_max="1" property_id="1" type_property="ATTACK"
  description="IPv4 address conflict detection (RFC5227). Possible ARP poisoning.">
  <operator value="THEN" delay_max="1">
    <event value="COMPUTE" event_id="1"
      description="ARP who has request"
      boolean_expression="(ARP.OPCODE == 1)"/>
  </operator>
</property>
```

³ The syntax and structure of MMT-Security properties are given in D2.WP3 deliverable in section 5.1

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 44 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

```

<event value="COMPUTE" event_id="2"
  description="ARP reply MAC address"
  boolean_expression="((ARP.OPCODE == 2)&&&
    (ARP.SRC_PROTO == ARP.DST_PROTO.1))"/>
</operator>
<event value="COMPUTE" event_id="3"
  description="ARP reply but with different MAC address"
  boolean_expression="(((ARP.OPCODE == 2)&&&
    (ARP.SRC_PROTO == ARP.DST_PROTO.1))&&&
    (ARP.SRC_HARD == ARP.SRC_HARD.2))"/>
</property>

```

1.4.1.2.2 Thales radio protocols

Ad hoc networks are more exposed to security threats than traditional networks due to their mobility and open architecture characteristics. In addition, any dysfunction due to badly configured nodes can severely affect the network as all nodes participate in the routing task. For these reasons, it is important to check the validity of ad hoc protocols, to verify whether the running implementation conforms to its security specification and to detect security flows in the network. In the context of the DIAMONDS project, Thales proposed a case study dealing with radio protocols described in D1.W1 deliverable section 2.4 [71]. Based on a risk analysis of their protocols, they specified a set of security rules that the system under test needs to respect. In this section, we present one example related to RLC_CL_UNIT_DATA_ACK messages.

Threat: Deny of service by flooding of RLC_CL_UNIT_DATA_ACK messages

Security rule: A message RLC_CL_UNIT_DATA_ACK must be preceded with a message RLC_CL_UNIT_DATA_REQ that previously asked for an acknowledgement (R == 00010000) (correlation with the USER_TRANSACTION_ID)

This security rule can be specified as follows:

$$e_1 \xrightarrow{BEFORE, -1, 1} e_2$$

Where:

- Event 1 (e_1): RLC_CL_UNIT_DATA_ACK message
- Event 2 (e_2): RLC_CL_UNIT_DATA_REQ message that asked for acknowledgement

This security rule is specified using XML⁴ as follows:

```

<property value="THEN" delay_min="-1" property_id="1" type_property="SECURITY_RULE"
  description="A message RLC_CL_UNIT_DATA_ACK must be preceded with a message
  RLC_CL_UNIT_DATA_REQ that asked for acknowledgement (R == 1) (correlation with the
  USER_TRANSACTION_ID)">
  <event value="COMPUTE" event_id="1"
    description="RLC_CL_UNIT_DATA_ACK message"
    boolean_expression="((BASE.PROTO == 5152)&&&
      ((BASE.TIME_SLOT == BASE.TIME_SLOT.2)&&&
      (MSG_RLC_CL_UNIT_DATA_ACK.USER_TRANSACTION_ID ==
      MSG_RLC_CL_UNIT_DATA_REQ.USER_TRANSACTION_ID.2)))/>
  <event value="COMPUTE" event_id="2"
    description="RLC_CL_UNIT_DATA_REQ message that asked for acknowledgement"
    boolean_expression="((BASE.PROTO == 1056)&&&
      (MSG_RLC_CL_UNIT_DATA_REQ.QOS_R == 128))"/>
</property>

```

⁴ The syntax and structure of MMT-Security properties are given in D2.WP3 deliverable in section 5.1

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 45 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

1.4.2 Deep Packet/Flow Inspection

Deep Packet Inspection (DPI) is a networking technology process based on examining the header and non-header content of a packet by a system that is not an endpoint in the communication. When a packet arrives, each layer is fully parsed and inspected. DPI enables diverse operations such as advanced network management, improvement of network security functions, and monitoring of customer data traffic in order to, for instance, mediate its speed. The use of DPI makes it possible to find, identify, classify packets with specific data or code payloads that conventional packet filtering, which examines only packet headers, cannot detect. Deep Flow Inspection (DFI), which is an evolution of DPI, is a way to identify and classify traffic flows in a network.

In the context of DIAMONDS, DPI and DFI are used to help detect and tackle harmful traffic and security threats; and, to throttle or block undesired behaviours. We define a set of security properties for network traffic, at both control and user planes, to catch interesting events. Indeed, based on the defined security properties, we register the attributes to be extracted from the inspected packets and flows. These attributes are of three types:

- Real attributes: They can be directly extracted from the inspected packet. They correspond to a protocol field value.
- Virtual attributes: They are calculated within a flow. Packets from the same flow are grouped and security/performance indicators are calculated (e.g. delays, jitter, packet loss rate) and made available to security analysis engine.
- Meta attributes: These attributes are linked to each packet to describe capture information. The capture time of each packet. The timestamp attribute is the main meta-attribute in the current version of Montimage's monitoring technique.

When finding an interesting event, we report this information (e.g. give values of attributes) to higher level monitoring system for further treatment. Besides, legal aspects need to be considered including storing the information required by law and protecting the privacy of citizens and organisations. If one needs to handle personal information on individuals, a number of legal obligations to protect that information need to be respected. As a legally sanctioned official access to private communications, Lawful Interception is a security process in which a service provider or network operator collects and provides law enforcement officials with intercepted communications of private individuals or organizations.

1.4.3 Network Traffic Analysis Based on MMT-Security Properties

Security monitoring based on MMT-Security properties is performed with the assistance of a network sniffer to capture network traffic. The analysis can be done:

- Offline: The analysis is based on an already captured network trace in pcap format.
- Online: In this case, passive monitoring does not cause any traffic overhead in the network as active testing would. However, some processing performance issues could exist since huge amounts of collected data have to be processed while they are being collected.

Network traffic analysis for security follows four steps:

- The definition of the monitoring architecture: This architecture depends on the nature of the system under test and its deployment in the network. Capture engines (i.e. probes) are placed at relevant elements or links in the network to obtain real time visibility of the traffic to be analysed. In the case of a distributed architecture, local trace files are merged (based on event timestamps) to obtain a broader visibility of what is going on in the network.
- The description of the system security goals and attacks based on the MMT-security property format: The description specifies the security rules that the studied system has to respect and/or the security attacks that it has to avoid. This task can be done by an expert of the system under test that understands its security requirements in detail.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 46 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

- The security analysis: Based on the security property specification, the passive tester performs security analysis on the captured trace file to deduce verdicts for each property.
- Reaction: In case of a fail verdict, some reactions have to be undertaken in the network, based on previously defined security strategies, e.g. to block any malicious behaviour.

Passive monitoring can be coupled with DIAMONDS active testing and fuzzing techniques. It can be applied as part of the testing chain, e.g. after the execution of some security-oriented test cases, in order to collect network traffic and analyse it based on previously defined MMT-Security properties. Passive monitoring is complementary to DIAMONDS security testing techniques.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 47 of 134 Version: 1.0 Date : 26.09.2011 Status : Final Confid : Public
---	--	---

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 48 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

1.5 HORN LOGIC BASED SECURITY ANALYSIS – A DATA CENTRIC APPROACH

Passive testing or runtime monitoring approaches are usually propositional in nature, i.e. they represent inputs and outputs as part of a (usually) finite set of symbols or values accounting for the control part of the communication, allowing techniques from regular language theory to be used. Since modern, particularly security based protocols depend heavily on data exchange, some works extend the propositional approach by using a concept as “parameterized propositions”[65], where propositions are augmented with parameters (variables) of different domains. Such extension allows to successfully test simple data relations, however, as the number of parameters increases, it does it in detriment of succinctness and readability of formulas. It is the premise of the approach depicted in this section where the main functionalities of the protocol are contained in the data and not in the control part, a data-centric approach, i.e. a definition of security properties from the basis of data fields and their expected relations, provides an effective solution for passively testing network protocols. In order to clearly understand the herein proposed approach and experiment it on a simple example, let us first give couple of definitions.

1.5.1 Protocol Messages and Traces

1.5.1.1 Messages in network protocols

A message in a communication protocol is, using the most general view, a collection of data fields belonging to multiple domains. Data fields in messages, are usually either atomic, i.e. the information they provide comes from using their value as a unit (e.g. timestamp, name, port), or compound, i.e. they are composed of multiple elements (e.g. a URI sip:name@domain.org). Due to this, we also divide the types of possible domains in atomic, defined as sets of numeric or string values, or compound, as follows.

Definition 1. A compound value v of length $k > 0$, is defined by the set of pairs $v = \{ (l_i, v_i) \mid l_i \in L \wedge v_i \in D_i \cup \{\epsilon\}, i = 1 \dots k \}$, where $L = \{l_1, \dots, l_k\}$ is a predefined set of labels and D_i are data domains, not necessarily disjoint.

In a compound value, in each element (l, v) , the label l represents the functionality of the piece of data contained in v . The length of each compound value is fixed, but undefined values can be allowed by using ϵ (null value). A compound domain is then the set of all values with the same set of labels and domains.

Definition 2. Given L a set of labels of length k and $D_1, \dots, D_k$ a group of data domains (not necessarily disjoint), with $k > 0$. A compound domain C is identified by the tuple $\langle L, D_1, \dots, D_k \rangle$. Each element $v \in C$ is a compound value of length k with labels $l_i \in L$ and corresponding $v_i \in D_i \cup \{\epsilon\}$, for $i = 1 \dots k$.

Definition 3. Let $C = \langle L, D_1, \dots, D_k \rangle$ be a compound domain. Then a function $\delta_C : C \times L \rightarrow \bigcup_{i=1..k} D_i \cup \{\epsilon\}$ is defined, where given a compound value $v \in C$ and label l_i , then $\delta_C(v, l_i) = v_i$ where (l_i, v_i) is a pair in v .

Notice that, D_i being domains, they can also be either atomic or compound, allowing for recursive structures to be defined. Finally, given a network protocol P , a compound domain M_P can generally be defined, where the set of labels and element domains derive from the message format defined in the protocol specification. A message of a protocol P is any element $m \in M_P$, that is, a message is any value which is valid with respect to the protocol specification.

Example 1. A possible message for the SIP protocol, specified using the previous definition is:

$$m = \{ (method, 'INVITE'), (status, \epsilon), (from, 'john@b.org'), (to, 'paul@b.org'), (cseq, \{ (num, 10), (method, 'INVITE') \}) \}$$

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 49 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

representing an INVITE request (a call request) from john@b.org to paul@b.org. Notice that the value associated to the label *cseq* is also a compound value, $\{(num, 10), (method, 'INVITE')\}$.

Accessing data inside messages is a basic requirement for the current approach, and nested calls to the function δ of a domain, as in $\delta_{cseq}(\delta_M(m, cseq), method)$ to obtain the value for the label *method* inside the value associated with *cseq* in the example. We will use in the following, the notation $v.l_1.l_2 \dots l_n$ (selector variable) to represent that call as for instance $m.cseq.method$ for the example above.

1.5.1.2 Execution traces

A trace is a sequence of messages of the same domain (i.e. using the same protocol) containing the interactions of an entity of a network, called the point of observation (P.O), with one or more peers during an indeterminate period of time (the life of the P.O). Such definition makes a trace potentially infinite, however, testing of properties can only occur in a finite segment of the trace. A definition of trace and trace segment is provided below.

Definition 5. Given the domain of messages M_P for a protocol P . A trace is a sequence $\Gamma = m_1, m_2, \dots$ of potentially infinite length, where $m_i \in M_P$.

Definition 6. Given a trace $\Gamma = m_1, m_2, \dots$, a trace segment is any finite sub-sequence of Γ . The order relations $\{<, >\}$ are defined in a trace, where for $m, m' \in \rho$, $m < m'$, $\text{pos}(m) < \text{pos}(m')$ and $m > m'$, $\text{pos}(m) > \text{pos}(m')$ and $\text{pos}(m) = i$, the position of m in the trace ($i \in \{1, \dots, \text{len}(\rho)\}$).

As testing can only be performed in trace segments, *trace* will be used to refer to a trace segment unless explicitly stated.

1.5.2 Horn Logic Syntax and Semantics

A **syntax** based on Horn clauses is used to express the security properties. The syntax is closely related to that of the query language Datalog, described in [1], for deductive databases, however, extended to allow for message variables and temporal relations.

Formulas in this logic are expressed by the use of common terms and atoms, where:

A term $t ::= c \mid x \mid x.l_1.l_2 \dots l_n$ where c is a constant in some domain (e.g. a message in a trace), x is a variable, l represents a label, and $x.l_1.l_2 \dots l_n$ is called a selector variable.

An atom $A ::= p(\overbrace{t_1, \dots, t_k}^k) \mid t=t \mid t \neq t$ where p is a predicate of label p and arity k (i.e. there are k term arguments for predicate p).

In this logic, relations between terms and atoms are stated by the definition of clauses. A *clause* is an expression of the form $A_0 \leftarrow A_1 \wedge \dots \wedge A_n$, where A_0 , called the head of the clause, has the form $A_0 = p(t^*_1, \dots, t^*_k)$, where t^*_i are a restriction on terms for the head of the clause ($t^* = c \mid x$). The expression $A_1 \wedge \dots \wedge A_n$, is called the body of the clause, where A_i are atoms.

Finally a formula is defined by the following BNF:

$\phi ::= A_1 \wedge \dots \wedge A_n \mid \phi \rightarrow \phi \mid \forall_x \phi \mid \forall_{y>x} \phi \mid \forall_{y<x} \phi \mid \exists_x \phi \mid \exists_{y>x} \phi \mid \exists_{y<x} \phi$

Where

- The $\rightarrow$ operator indicates causality in a formula, and should be read as “if-then” relation.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 50 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

- The $\exists$ and $\forall$ quantifiers, are equivalent to its counterparts in predicate logic. However, as it will be seen on the semantics, here they only apply to messages in the trace. Then, for a trace ρ , $\forall x$ is equivalent to $\forall x \in \rho$ and $\forall y < x$ is equivalent to $\forall y \in \rho, y < x$, with ' $<$ ' indicating the order relation from Definition 6. These types of quantifiers are called trace quantifiers.

The semantics used in this approach is related to the traditional Apt–Van Emdem–Kowalsky semantics for logic programs [70], however some changes are introduced to deal with messages and trace temporal quantifiers. We begin by introducing the concepts of substitutions (as defined in [59]).

Definition 7. A substitution is a finite set of bindings $\theta = \{x_1/t_1, \dots, x_k/t_k\}$ where each t_i is a term and each x_i is a variable such that $x_i \neq t_i$ and $x_i \neq x_j$ if $i \neq j$.

The application $x\theta$ of a substitution θ to a variable x is defined by t if $x/t \in \theta$ and x otherwise. Its application to a selector variable $x.l_1 \dots l_k$ is defined as $t.l_1 \dots l_k$ if $x/t \in \theta$ with t a compound value and $t.l_1 \dots l_k$ otherwise.

Given $K = \{C_1, \dots, C_p\}$ a set of clauses and $\rho = m_1, \dots, m_n$ a trace. An interpretation is any function I mapping an expression E that can be formed with elements (clauses, atoms, terms) of K and terms from ρ to one element of $\{\perp, T\}$. It is said that E is true in I if $I(E) = T$.

The **semantics of formulas** under a particular interpretation I , is given by the following rules.

- The expression $t_1 = t_2$ is true, iff t_1 equals t_2 (they are the same term).
- The expression $t_1 \neq t_2$ is true, iff t_1 is not equal to t_2 (they are not the same term).
- An atom $A = p(c_1, \dots, c_k)$ is true, iff $A \in I$.
- An atom A is true, iff every instance of A is true in I .
- The expression $A_1 \wedge \dots \wedge A_n$, where A_i are atoms, is true, iff every A_i is true in I .
- A clause $C: A_0 \leftarrow B$ is true, iff every instance of C is true in I .
- A set of clauses $K = \{C_1, \dots, C_p\}$ is true, iff every clause C_i is true in I .

An interpretation is called a model for a clause set $K = \{C_1, \dots, C_p\}$ and a trace ρ if every $C_i \in K$ is true in I . A formula ϕ is true for a set K and a trace ρ (true in K, ρ for short), if it is true in every model of K, ρ . The semantics of formulas is then defined as follows. Let K be a clause set, ρ a trace for a protocol and M a model, the operator $\hat{M}$ defines the semantics of formulas.

$$\hat{M}(A_1 \wedge \dots \wedge A_n) = \begin{cases} T & \text{if } M(A_1 \wedge \dots \wedge A_n) = T \\ \perp & \text{otherwise} \end{cases}$$

The semantics for trace quantifiers requires first the introduction of a new truth value '?' (inconclusive) indicating that no definite response can be provided. The semantics of quantifiers $\forall$ and $\exists$ is defined as follows:

$$\hat{M}(\forall_x \phi) = \begin{cases} \perp & \text{if } \exists \theta \text{ with } x/m \in \theta \text{ and } m \in \rho, \\ & \text{where } \hat{M}(\phi\theta) = \perp \\ ? & \text{otherwise} \end{cases}$$

$$\hat{M}(\exists_x \phi) = \begin{cases} T & \text{if } \exists \theta \text{ with } x/m \in \theta \text{ and } m \in \rho, \\ & \text{where } \hat{M}(\phi\theta) = T \\ ? & \text{otherwise} \end{cases}$$

Since ρ is a finite segment of an infinite execution, it is not possible to declare a 'T' result for $\forall_x \phi$, since we do not know if ϕ may become ' $\perp$ ' after the end of ρ . Similarly, for $\exists_x \phi$, it is unknown whether ϕ becomes true in the future. This issue is further detailed on the technical report [31]. The rest of the quantifiers are detailed in the following, where x is assumed to be bound as a message previously obtained by $\forall x$ or $\exists x$.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 51 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

$$\hat{M}(\forall_{y>x}\phi) = \begin{cases} \perp & \text{if } \exists\theta \text{ with } y/m \in \theta, \\ & \text{where } \hat{M}(\phi\theta) = \perp \text{ and } m > x \\ ? & \text{otherwise} \end{cases}$$

$$\hat{M}(\exists_{y>x}\phi) = \begin{cases} \top & \text{if } \exists\theta \text{ with } y/m \in \theta, \\ & \text{where } \hat{M}(\phi\theta) = \top \text{ and } m > x \\ ? & \text{otherwise} \end{cases}$$

The semantics for $\forall y < x$ and $\exists y < x$ is equivalent to the last two formulas, exchanging $>$ by $<$.

1.5.3 Security Properties Evaluation Algorithm on Execution Traces

In the previous section, a two part semantic was defined for the logic: one for formulas of type $A1 \wedge \dots \wedge A_n$ (atomic formulas), and a second one for formulas including trace quantifiers. Thus an evaluation algorithm is required for security properties with a two part methodology: 1) resolution of atomic formulas, where a variant of the classical SLD (Selective Linear Definite-clause) resolution algorithm [11] can be performed. 2) Evaluation of quantifiers and declaration of verdicts for a given trace.

Since the SLD resolution algorithm is widely covered in the literature, we will not detail the application and will only focus on the evaluation of formulas in traces.

Two rules are used to report the results of the evaluation of a property in a trace:

1. Given a formula $\forall x\phi$, an independent result should be declared for every value of x .
2. Given a formula $\exists x\phi$, a result should be given only if it exist some values of x in the trace that makes the property true. Any other values for x are irrelevant for the resolution.

An algorithm can be constructed using recursion and the details of the different cases in the evaluation are provided as follows, where $eval(\phi, \theta, \rho)$ returns the evaluation of the formula ϕ using substitution θ into trace ρ .

- Evaluation of a formula $\forall x\phi$ provides an independent solution with the evaluation of ϕ for every possible value of x in the trace. The result is provided by the following formula.

$$eval(\forall_x\phi, \theta, \rho) = \begin{cases} eval(\phi, \alpha, \rho) & \forall m \in \rho \text{ with} \\ & \alpha = \theta \cup \{y/m\} \\ ? & \text{if no } \perp \text{ results} \\ & \text{were found} \end{cases}$$

- Evaluation of a formula $\exists x$ looks for a 'T' result to the evaluation of ϕ .

$$eval(\exists_x\phi, \theta, \rho) = \begin{cases} \top & \text{if } \exists m \in \rho \text{ where} \\ & eval(\phi, \alpha, \rho) = \top \text{ with} \\ & \alpha = \theta \cup \{x/m\} \\ ? & \text{otherwise} \end{cases}$$

- Evaluation of a formula $\forall y > x\phi$ assumes that a binding for x to a message in the trace already exists in θ and provides a solution for every evaluation of ϕ after the position of that message. Depending on the implementation, an error should occur if no previous binding for x exists.

$$eval(\forall_{y>x}\phi, \theta, \rho) = \begin{cases} eval(\phi, \alpha, \rho) & \forall m \in \rho \text{ with} \\ & m > x\theta \text{ and} \\ & \alpha = \theta \cup \{x/m\} \\ ? & \text{if no } \perp \text{ results} \\ & \text{were found} \end{cases}$$

- Evaluation of a formula $\exists y > x$ looks for a 'T' result to the evaluation of ϕ after the position of x in the substitution θ .

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 52 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

$$eval(\exists_{y>x}\phi, \theta, \rho) = \begin{cases} \top & \text{if } \exists m \in \rho \text{ with } m > x\theta \\ & \text{where } eval(\phi, \alpha, \rho) = \top, \\ & \alpha = \theta \cup \{x/m\} \\ ? & \text{otherwise} \end{cases}$$

- Evaluation of $\forall y < x \phi$ and $\exists y < x \phi$ are analogous to their equivalents with ' $>$ ' just replacing every occurrence of ' $>$ ' by ' $<$ '.
- Evaluation of a formula $\phi \rightarrow \Psi$ first will evaluate ϕ and if the result is ' $\top$ ', then evaluate Ψ . If the latter also has the value ' $\top$ ', then the result of evaluation is ' $\top$ '. Any new bindings defined from the evaluation of ϕ must be used in the evaluation of Ψ . If the evaluation of ϕ is ' $\perp$ ', then the result is ignored, since $\perp \rightarrow \Psi$ is not an interesting result because it is independent of the value of Ψ .

$$eval(\phi \rightarrow \psi, \theta, \rho) = \begin{cases} \top & \text{if } eval(\phi, \theta, \rho) = \top \text{ and} \\ & eval(\psi, \theta, \rho) = \top \\ \perp & \text{if } eval(\phi, \theta, \rho) = \top \text{ and} \\ & eval(\psi, \theta, \rho) = \perp \\ ? & \text{if } eval(\phi, \theta, \rho) = ? \text{ or} \\ & eval(\psi, \theta, \rho) = ? \end{cases}$$

- Evaluation of a formula $A_1 \wedge \dots \wedge A_k$, where A_i are atoms, returns the value obtained using SLD-resolution.

$$eval(A_1 \wedge \dots \wedge A_k, \theta, \rho) = \begin{cases} \top & \text{if } (A_1 \wedge \dots \wedge A_k)\theta \\ & \text{has a solution} \\ \perp & \text{otherwise} \end{cases}$$

The previous rules define every possible case for the algorithm evaluating functional, non-functional or security properties on an execution trace.

1.5.4 Examples and Experiments

The Session Initiation Protocol (SIP) [65] is an application-layer protocol that relies on request and response messages for communication, and it is an essential part for communication within the IMS (IP Multimedia Subsystem) framework. Messages contain a header which provides session, service and routing information, as well as an (optional) body part to complement or extend the header information. Several RFCs have been defined to extend the protocol with to allow messaging, event publishing, notification and security. These extensions are used by services of the IMS such as the Presence service [4] and the Push-to-talk Over Cellular (PoC) service [9].

For the experiments, traces for an ad-hoc PoC session establishment were obtained from a production IMS implementation, provided by Alcatel-Lucent France and extracted from the interfaces of the IMS core, containing exchange between the client and the IMS presence and PoC services. Tests were performed using a prototype implementation of the concepts presented in the current work. The implementation, defined properties and network traces can be found in <http://www-public.int-evry.fr/~lalanne/datamon.html>.

Properties, extracted from the requirements for the PoC and Presence service were evaluated in the traces and the approach showed successful in determining the satisfaction of such properties. Three of the properties tested are provided in the following, where the structure of SIP messages used is described by the following table.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 53 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Label	Description
method	Indicates the request type, i.e. REGISTER, INVITE, etc, from the first line in a SIP request message. If empty (ϵ) then message is a response.
statusCode	Numeric code for the status, obtained from first line in a response message. If empty then the message is a request.
from	URI indicating the sender of the message.
to	URI indicating the recipient of the message.
callId	Unique identifier to group series of messages.
cSeq	Identifies the transaction the message belongs to with a sequence and a transaction originating method.
cSeq.seq	Sequence for the transaction.
cSeq.method	Method that originated the transaction

1. **For every request there must be a response.** This property can be used for a monitoring purpose, in order to draw further conclusions from the results. Due to the issues relative to testing on finite traces for infinite executions, false results can never be given for this context. However, inconclusive results can be returned and conclusions may require further analysis of the results (for instance if the same type of message is always without a response). The property evaluated is as follows:

$$\forall_x (\text{request}(x) \wedge x.\text{method} \neq \text{'ACK'} \rightarrow \exists_{y>x} (\text{nonProvisional}(y) \wedge \text{responds}(y, x)))$$

where $\text{nonProvisional}(x)$ accepts all non-provisional responses (non-final responses, with $\text{status} \geq 200$) to requests with method different than ACK, which does not require a response. The predicate $\text{responds}(y; x)$ accepts all pairs of messages ($y; x$) where y is a response to x and is defined by:

$$\begin{aligned} \text{responds}(x, y) \leftarrow & \text{response}(x) \\ & \wedge x.\text{from} = y.\text{from} \wedge x.\text{to} = y.\text{to} \\ & \wedge x.\text{callId} = y.\text{callId} \wedge x.\text{cSeq.seq} = y.\text{cSeq.seq} \\ & \wedge x.\text{cSeq.method} = y.\text{cSeq.method} \end{aligned}$$

2. **No session can be initiated without a previous registration.** This property can be used to test that only users successfully registered with the SIP Core can initiate a PoC session (or a SIP call, depending on the service). It is defined using our syntax as follows:

$$\forall_x (\exists_{y>x} \text{sessionEstablished}(x, y) \rightarrow \exists_{u<x} (\exists_{v>u} \text{registration}(u, v)))$$

where $\text{sessionEstablished}$ and registration are defined as

$$\begin{aligned} \text{sessionEstablished}(x, y) \leftarrow & x.\text{method} = \text{'INVITE'} \\ & \wedge y.\text{statusCode} = 200 \\ & \wedge \text{responds}(y, x) \end{aligned}$$

$$\begin{aligned} \text{registration}(x, y) \leftarrow & \text{request}(x) \wedge \text{responds}(y, x) \\ & \wedge x.\text{method} = \text{'REGISTER'} \\ & \wedge y.\text{statusCode} = 200 \end{aligned}$$

3. **Subscription to events and notifications.** The presence service is a system to disseminate presence information and relies on SIP for communication. There, a user (the watcher) can subscribe to be notified of another user's (the presentity) presence information, this works by using the SIP messages SUBSCRIBE, PUBLISH and NOTIFY for subscription, update and notification respectively. It is desirable then to test, that whenever there is a subscription, a notification MUST occur upon an update event. This can be tested with the following formula:

$$\begin{aligned} \forall_x (\text{update}(x, \text{user}, \text{event}) \\ \rightarrow (\exists_{y<x} \text{subscribe}(y, \text{watcher}, \text{user}, \text{event}) \\ \rightarrow \exists_{z>x} \text{notify}(z, \text{watcher}, \text{user}, \text{event}))) \end{aligned}$$

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 54 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

where *subscribe*, *update* and *notify* hold on SUBSCRIBE, PUBLISH and NOTIFY events respectively. Notice that the values of the variables *watcher*, *user* and *event* may not have a value at the beginning of the evaluation, in that case their value is set by the evaluation of the subscribe clause, as shown in the following formula, where '=' works as comparison or assignment depending on whether the variables have a previous binding.

$$\begin{aligned}
 & \text{subscribe}(x, \text{watcher}, \text{user}, \text{event}) \\
 & \leftarrow x.\text{method} = \text{'SUBSCRIBE'} \\
 & \quad \wedge \text{watcher} = x.\text{from} \\
 & \quad \wedge \text{user} = x.\text{to} \\
 & \quad \wedge \text{event} = x.\text{event}
 \end{aligned}$$

The results of the experiments are shown in Table below. A predominance of inconclusive results were found. In the second property, for example, inconclusive results indicate that a session initialization was found on the trace, but no registration procedure was captured. If prior to testing, the assumption is made, due to the trace capturing methodology, that a session registration should be available, then such inconclusive results are telling and indicate a possible fault in the implementation. Unfortunately, such assumption could not be made in our case.

Trace	Messages	Property 1				Property 2				Property 3			
		T	⊥	?	Time(s)	T	⊥	?	Time(s)	T	⊥	?	Time(s)
1	31	6	0	0	0.556	0	0	0	0.744	0	0	0	0.416
2	62	24	0	0	0.552	0	0	4	1.13	0	0	0	0.312
3	126	48	0	0	0.423	0	0	8	2.726	0	0	0	0.609
4	141	55	0	9	0.48	0	0	6	1.869	0	0	0	0.365
5	189	99	0	0	0.809	0	0	0	1.714	0	0	0	0.38
6	190	78	0	0	0.504	0	0	6	2.851	0	0	0	0.273
7	214	93	0	0	0.352	0	0	8	4.494	0	0	0	0.338
8	331	151	0	0	0.699	0	0	0	4.588	0	0	0	0.272
9	409	206	0	0	0.985	0	0	4	10.155	4	0	0	0.479
10	625	281	0	14	1.457	4	0	0	38.874	4	0	0	0.563

1.6 MACHINE LEARNING

1.6.1 Introduction

Machine learning approaches are used widely in different areas of science and information security is one of those areas. Machine learning gives a good set of tools in 1) identifying limited amount of behaviours defined by measurement combinations or 2) detecting abnormal behaviour when normal behaviour of the system is well defined.

Different machine learning techniques are many, with a textbook⁵ on artificial intelligence listing over ten with dozens of modifications available for each approach. Each technique has its intrinsic set of strengths and weaknesses. The approaches have enjoyed a storm of successes reported in scientific papers. Especially K-means clustering and SVM approaches have seen great amount of usage.

1.6.2 Challenges

In their paper⁶ discussing problems of machine learning in intrusion detection approaches Sommer and Paxson discuss in length multiple problems that machine learning application faces when used in real world network situation. The authors specifically identify the following list of difficulties that the machine learning approach needs to account for in real world network monitoring application, but many of them can be generalised to different information security problems.

⁵ S. Russel and P. Norvig, "Artificial Intelligence, A Modern Approach", 2nd Edition, Prentice Hall, 2003

⁶ R. Sommer and V. Paxson, "Outside the Closed World: On Using Machine Learning for Network Intrusion Detection", Proc. IEEE Symposium on Security and Privacy, May 2010

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 55 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

- Outlier detection
- High cost of errors
- Semantic gap
- Diversity of network traffic
- Difficulties with evaluation

1.6.2.1 Outlier detection

Machine-learning algorithms excel at finding similarities, while detection of abnormal activity proves more difficult. Machine learning classification needs typically to be trained to detect the required classes by using samples which describe measurements from all classes. This training material is not available for novel attacks, or novel variations of new attacks.

The problem is well described in a paper⁷ by Witten *et. al*, where the authors argued that by specifying only positive examples and adopting a standing assumption that the rest are negative is called the closed world assumption. Real world problems rarely involve closed worlds and information security even less so.

1.6.2.2 High cost of errors

The relative cost of errors in information security systems, especially intrusion detection, has limited their real world applicability. False positive matches require expensive analysis by a security analyst and false negatives can cause massive costs from administration work only (e.g. purging malware from systems).

1.6.2.3 Semantic gap

When using a machine learning system to spot suspicious activity for a human operator, semantic gap becomes a problem. The question “What does the operator need to do?” is not answered by machine learning algorithms, which just provide a classification or alert. Providing the input metrics for the analyst can also be confusing and counterproductive. As machine learning systems can easily incorporate tens of input metrics, it is important to provide the data in a careful manner, for the operator to be able to interpret what has happened. Visualisation and linking the data become important, as well as understanding the environment. This is especially problematic, when many false positives eat away at the attention of the operator, who can be easily overwhelmed by the events and data.

1.6.2.4 Diversity of network traffic

The diversity of network traffic in unrestricted environment often surprises people without first hand experience on managing different networks. There can be an immense variety in even the basic characteristics making network behaviour highly unpredictable. This makes the notion of normality elusive, which in turn makes detection of security events unreliable, as normal variance can easily trigger false alarms.

Outside network security the same problems are likely when the systems behaviour is hard to predict (e.g. analysis of unknown programs maliciousness). This needs to be kept in mind when thinking of using machine learning approach for detection interesting behaviour.

1.6.2.5 Difficulties with evaluation

The data, which the machine learning algorithm is trained and tested can be problematic. Especially network traffic lacks public data, which could be used for realistic training and testing of such systems. This causes the researchers often to make their own training and evaluation samples, which can be highly simplified. Any results that the system gains do not often apply in “live” environment.

The information security machine learning system is also located in a competitive setting. The defenders and attackers are locked in invisible arms-race to create better defences and attacks to penetrate those. This can easily reduce the effectiveness of any widely available system.

⁷ I. H. Witten and E. Frank, “Data Mining: Practical Machine Learning Tools and Techniques” 2nd edition, Morgan Kaufmann, 2005

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 56 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

1.6.3 Research

Great amount of papers can be found using any major scientific collections on machine learning and information security. This part will discuss only few in order to indicate the wide range of problems within information security, that the machine learning approaches have applied to.

“Machine Learning Techniques for Passive Network Inventory”⁸, by Francois *et. al.* describe a network fingerprinting technique taking advantage of SVM algorithm to classify devices.

“Neural Network Techniques for Proactive Password Checking”⁹, by Ciaramella *et. al* describe techniques to use neural networks for enforcing password strength requirements.

“A Plan for Spam”¹⁰ by Graham describes a machine learning algorithm used for spam email detection and filtering. This has been one of the few machine learning applications in information security area which has seen wide usage.

“Evolutionary Neural Networks for Anomaly Detection Based on the Behaviour of a Program”¹¹ by Han *et. al* describe host based intrusion detection system which gives impressive accuracy for detection of attacks against the system. The impressive results however have not led to wide scale adaptation of their detection scheme.

Many other papers can be found tackling different aspects of information security in conferences and journals around the world, with new papers joining the ranks each month. Machine learning has much to give, when applied correctly.

1.7 BINARY CODE INSTRUMENTATION

1.7.1 Introduction

Binary instrumentation deals with the capability to monitor the operation of software. Its main aim is to observe the dynamic program behaviour without affecting its operation. In order to reach this goal it adds specific instructions to monitor and log the activities and interacts with the kernel to observe the activity of a program, as system calls execution, networks or sockets operation, file system access...

This approach is required during a security test to clearly identify the origin of an anomaly or a crash. All these probes improve the detection of functions generating errors and provide information about their arguments and their specific contexts.

This specific approach of software engineering can be divided in two parts: Tracing and debugging. The concepts of these two parts are described in the followings paragraphs.

1.7.2 Tracing

Tracing involves either the activation of some probes or the injection of some codes, in order to monitor program execution.

⁸ J. Francois, H. Abdelnur, R. State and O. Fester: “Machine Learning Techniques for Passive Network Inventory”, IEEE Transactions on Network and Service Management, December 2010

⁹ A. Ciaramella, P. D’Arco, A. De Santis, C. Galdi and R. Tagliaferri: “Neural Network Techniques for Proactive Password Checking”, IEEE Transactions on Dependable and Secure Computing, 2006

¹⁰ P. Graham: “A Plan for Spam”, Hackers & Painters, O’Reilly, 2004

¹¹ S.J. Han and S.B. Cho: “Evolutionary Neural Networks for Anomaly Detection Based on the Behaviour of a Program”, IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, June 2005

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 57 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

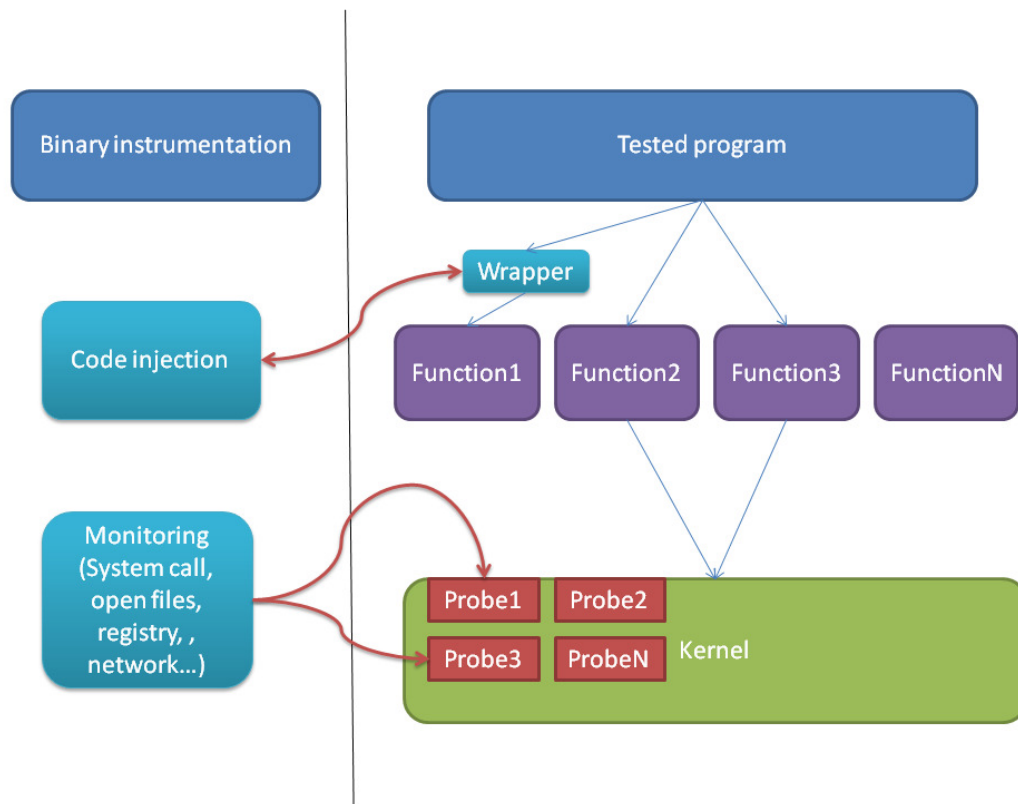


Figure 10: Tracing schema

In this part of the document, we deal on the one hand with different types of probes such as “system call monitoring”, “open files monitoring”, “registry monitoring”, etc..., and on the other hand with code injection (statically and dynamically).

1.7.2.1 System call monitoring

When a program makes a request to the kernel, it uses a system call. A system call is used when a program wants to interact with hardware, execute another process, communicate with kernel services, etc. System calls provide the interface between a process and the operating system. Monitoring these calls is interesting to understand what the program does.

An operating system natively allows monitoring system calls. Indeed, it ensures/enables to easily check open files, open sockets or binary execution. For example, we can use `strace` on Linux to list system calls made by the program:

```
open(".", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
fcntl64(3, F_GETFD) = 0x1 (flags FD_CLOEXEC)
getdents64(3, /* 18 entries */, 4096) = 496
getdents64(3, /* 0 entries */, 4096) = 0
close(3) = 0
fstat64(1, {st_mode=S_IFIFO|0600, st_size=0, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7f2c000
write(1, "autofs\nbackups\ncache\nflexlm\ngames"... , 86autofsA
```

Figure 11: Example of `strace` output on Linux

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 58 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

This approach may identify bugs as permission denied on `open()` function during a fuzzing test. The same kind of tools exists on Windows, for example, API Monitor:

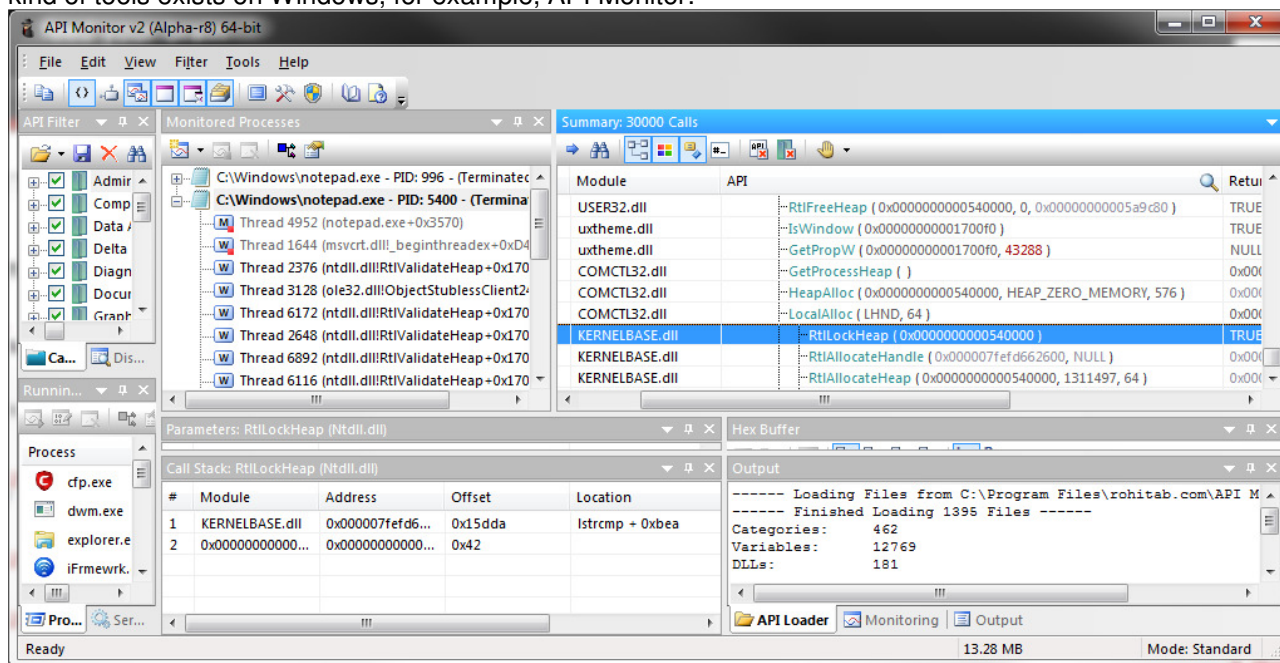


Figure 12: API Monitor interface on Windows

On UNIX, we can use a specific kernel API (Application Programming Interface) to monitor system calls directly through the kernel. On Solaris or FreeBSD, it is called *DTrace* and on *Linux SystemTap*. The advantage of this technique compared with the previous tools, is that it is not simply a tool, but it is a complete scripting language. It is much more flexible and allows more functionalities. For example, the following DTrace's one-line script displays files opened by a process:

```
# Files opened by process,
dtrace -n 'syscall::open*:entry { printf("%s %s",execname,copyinstr(arg0)); }'
```

Figure 13: DTrace example

The following SystemTap example allows printing system calls count for each process, one by one:

```
global syscalls

probe begin {
    print ("Collecting data... Type Ctrl-C to exit and display results\n")
}

probe syscall.* {
    syscalls[execname()]++
}

probe end {
    printf ("%10s %-s\n", "#SysCalls", "Process Name")
    foreach (proc in syscalls-)
        printf ("%10d %-s\n", syscalls[proc], proc)
}
```

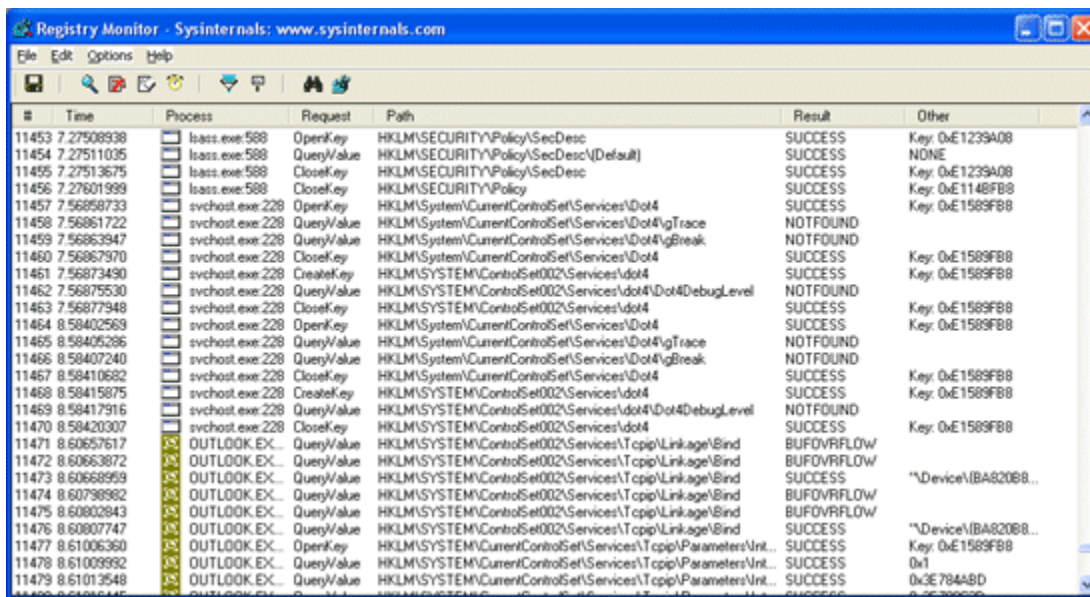
Figure 14: SystemTap example

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 59 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

1.7.2.2 Registry monitoring

The Windows Registry is a hierarchical database that stores configuration settings and options on Microsoft Windows operating systems. It is interesting to monitor this activity because it permits to analyse what the program does.

We can use RegMon from Sysinternals toolbox to monitor Windows Registry activities:



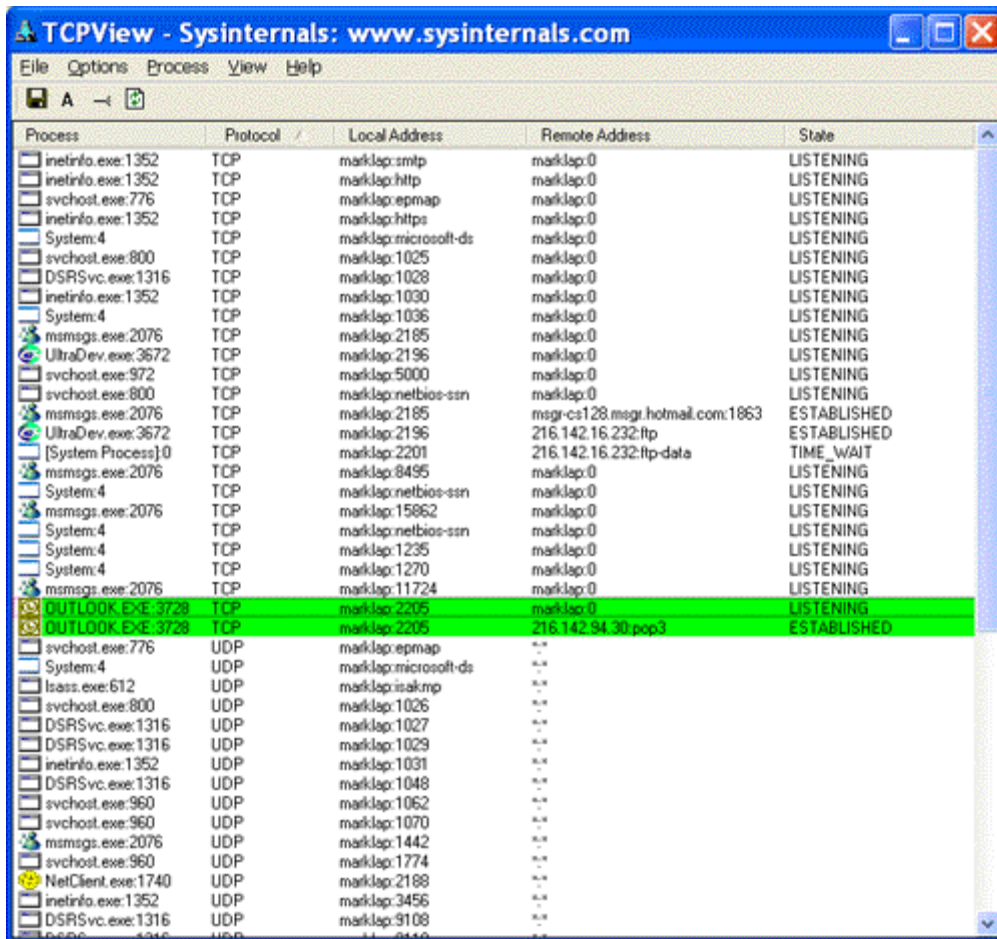
Time	Process	Request	Path	Result	Other
11453 7.27508938	lsass.exe:588	OpenKey	HKLM\SECURITY\Policy\SecDesc	SUCCESS	Key: 0xE1239A08
11454 7.27511035	lsass.exe:588	QueryValue	HKLM\SECURITY\Policy\SecDesc(Default)	SUCCESS	NONE
11455 7.27513675	lsass.exe:588	CloseKey	HKLM\SECURITY\Policy\SecDesc	SUCCESS	Key: 0xE1239A08
11456 7.27601939	lsass.exe:588	CloseKey	HKLM\SECURITY\Policy\SecDesc	SUCCESS	Key: 0xE1148FB8
11457 7.56858733	svchost.exe:228	OpenKey	HKLM\System\CurrentControlSet\Services\Dot4	SUCCESS	Key: 0xE1589FB8
11458 7.56861722	svchost.exe:228	QueryValue	HKLM\System\CurrentControlSet\Services\Dot4\gTrace	NOTFOUND	
11459 7.56863947	svchost.exe:228	QueryValue	HKLM\System\CurrentControlSet\Services\Dot4\gBreak	NOTFOUND	
11460 7.56867970	svchost.exe:228	CloseKey	HKLM\System\CurrentControlSet\Services\Dot4	SUCCESS	Key: 0xE1589FB8
11461 7.56873490	svchost.exe:228	CreateKey	HKLM\SYSTEM\ControlSet002\Services\dot4	SUCCESS	Key: 0xE1589FB8
11462 7.56875530	svchost.exe:228	QueryValue	HKLM\SYSTEM\ControlSet002\Services\dot4\Dot4DebugLevel	NOTFOUND	
11463 7.56877948	svchost.exe:228	CloseKey	HKLM\SYSTEM\ControlSet002\Services\dot4	SUCCESS	Key: 0xE1589FB8
11464 8.58402563	svchost.exe:228	OpenKey	HKLM\System\CurrentControlSet\Services\Dot4	SUCCESS	Key: 0xE1589FB8
11465 8.58405286	svchost.exe:228	QueryValue	HKLM\System\CurrentControlSet\Services\Dot4\gTrace	NOTFOUND	
11466 8.58407240	svchost.exe:228	QueryValue	HKLM\System\CurrentControlSet\Services\Dot4\gBreak	NOTFOUND	
11467 8.58410682	svchost.exe:228	CloseKey	HKLM\System\CurrentControlSet\Services\Dot4	SUCCESS	Key: 0xE1589FB8
11468 8.58415875	svchost.exe:228	CreateKey	HKLM\SYSTEM\ControlSet002\Services\dot4	SUCCESS	Key: 0xE1589FB8
11469 8.58417916	svchost.exe:228	QueryValue	HKLM\SYSTEM\ControlSet002\Services\dot4\Dot4DebugLevel	NOTFOUND	
11470 8.58420307	svchost.exe:228	CloseKey	HKLM\SYSTEM\ControlSet002\Services\dot4	SUCCESS	Key: 0xE1589FB8
11471 8.60657617	OUTLOOK.EX...	QueryValue	HKLM\SYSTEM\ControlSet002\Services\Tcpip\Linkage\Bind	BUFDVRFLOW	
11472 8.60663872	OUTLOOK.EX...	QueryValue	HKLM\SYSTEM\ControlSet002\Services\Tcpip\Linkage\Bind	BUFDVRFLOW	
11473 8.60668969	OUTLOOK.EX...	QueryValue	HKLM\SYSTEM\ControlSet002\Services\Tcpip\Linkage\Bind	SUCCESS	"Device\BA820B8...
11474 8.60798982	OUTLOOK.EX...	QueryValue	HKLM\SYSTEM\ControlSet002\Services\Tcpip\Linkage\Bind	BUFDVRFLOW	
11475 8.60802943	OUTLOOK.EX...	QueryValue	HKLM\SYSTEM\ControlSet002\Services\Tcpip\Linkage\Bind	BUFDVRFLOW	
11476 8.60807747	OUTLOOK.EX...	QueryValue	HKLM\SYSTEM\ControlSet002\Services\Tcpip\Linkage\Bind	SUCCESS	"Device\BA820B8...
11477 8.61006360	OUTLOOK.EX...	OpenKey	HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\Un...	SUCCESS	Key: 0xE1589FB8
11478 8.61009392	OUTLOOK.EX...	QueryValue	HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\Un...	SUCCESS	0x1
11479 8.61013548	OUTLOOK.EX...	QueryValue	HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\Un...	SUCCESS	0xE784ABD

Figure 15 : RegMon interface on Windows

1.7.2.3 Network monitoring

When a program wants to communicate through networks, it needs to open a network socket. It is a bidirectional inter-process communication flow across an Internet Protocol-based computer network. Socket manipulation is an API provided by the operating system. Monitoring these sockets allow the engineer to visualise network connections established or initialised by the program. During a fuzzing test, it allows monitoring the correct behaviour of these calls.

On Linux and Windows, we can use `netstat` to monitor all open/established/closed/waiting sockets. We can also use system calls monitoring method as described in section 1.7.2.1, but there exist some specific tools as TCPView from Sysinternals:



Process	Protocol	Local Address	Remote Address	State
inetinfo.exe:1352	TCP	marklap:smtp	marklap:0	LISTENING
inetinfo.exe:1352	TCP	marklap:http	marklap:0	LISTENING
svchost.exe:776	TCP	marklap:epmap	marklap:0	LISTENING
inetinfo.exe:1352	TCP	marklap:https	marklap:0	LISTENING
System:4	TCP	marklap:microsoft-ds	marklap:0	LISTENING
svchost.exe:800	TCP	marklap:1025	marklap:0	LISTENING
DSRSvc.exe:1316	TCP	marklap:1028	marklap:0	LISTENING
inetinfo.exe:1352	TCP	marklap:1030	marklap:0	LISTENING
System:4	TCP	marklap:1036	marklap:0	LISTENING
msmsgs.exe:2076	TCP	marklap:2185	marklap:0	LISTENING
UltraDev.exe:3672	TCP	marklap:2196	marklap:0	LISTENING
svchost.exe:972	TCP	marklap:5000	marklap:0	LISTENING
svchost.exe:800	TCP	marklap:netbios-ssn	marklap:0	LISTENING
msmsgs.exe:2076	TCP	marklap:2185	msgr-cs128.msgs.hotmail.com:1863	ESTABLISHED
UltraDev.exe:3672	TCP	marklap:2196	216.142.16.232:ftp	ESTABLISHED
[System Process] 0	TCP	marklap:2201	216.142.16.232:ftp-data	TIME_WAIT
msmsgs.exe:2076	TCP	marklap:8495	marklap:0	LISTENING
System:4	TCP	marklap:netbios-ssn	marklap:0	LISTENING
msmsgs.exe:2076	TCP	marklap:15862	marklap:0	LISTENING
System:4	TCP	marklap:netbios-ssn	marklap:0	LISTENING
System:4	TCP	marklap:1235	marklap:0	LISTENING
System:4	TCP	marklap:1270	marklap:0	LISTENING
msmsgs.exe:2076	TCP	marklap:11724	marklap:0	LISTENING
OUTLOOK.EXE:3728	TCP	marklap:2205	marklap:0	LISTENING
OUTLOOK.EXE:3728	TCP	marklap:2205	216.142.94.30:pop3	ESTABLISHED
svchost.exe:776	UDP	marklap:epmap	...	...
System:4	UDP	marklap:microsoft-ds	...	...
lsass.exe:612	UDP	marklap:isakmp	...	...
svchost.exe:800	UDP	marklap:1026	...	...
DSRSvc.exe:1316	UDP	marklap:1027	...	...
DSRSvc.exe:1316	UDP	marklap:1029	...	...
inetinfo.exe:1352	UDP	marklap:1031	...	...
DSRSvc.exe:1316	UDP	marklap:1048	...	...
svchost.exe:960	UDP	marklap:1062	...	...
svchost.exe:960	UDP	marklap:1070	...	...
msmsgs.exe:2076	UDP	marklap:1442	...	...
svchost.exe:960	UDP	marklap:1774	...	...
NetClient.exe:1740	UDP	marklap:2188	...	...
inetinfo.exe:1352	UDP	marklap:3456	...	...
DSRSvc.exe:1316	UDP	marklap:3108	...	...

Figure 16: TCPView output on Windows

1.7.2.4 Open files monitoring

It might be necessary to monitor files and disk operations. On Windows operating systems, Sysinternals provides a lot of tools to monitor this, for example AccessChk, DiskExt, DiskMon...

These tools can be used to analyse if no disk or file access modification is performed during the fuzzing test. We can also use system calls monitoring to specifically check disk or file system activities.

1.7.2.5 Code injection monitoring

Code injection is a technique used by hackers to run arbitrary code through a running process. For example, on Windows operating systems, DLL injection is a technique to add our own libraries, replace official libraries and take control of an application or spy the user.

A rootkit is a software that enables continued privileged access to a jeopardised computer. It hides its presence from administrators and injects code by replacing kernel system calls. For example, the rootkit replaces the function which lists all running processes to one which never shows the rootkit name. This it is not only useful for hackers; it can also be used by developers to monitor their program.

On Linux, we can use a special variable to overwrite native libraries: LD_PRELOAD. If we set this variable to a custom library, the program will use it.

```
# cat soft.c
#include <stdio.h>

int main()
```

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 61 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

```
{
    printf("pid : %u\n",getpid());
    return(0);
}
# gcc soft.c -o soft
# ./soft
pid : 43562
```

Figure 17: Simple getpid() example

Now create our own getpid() library and set LD_PRELOAD:

```
# cat getpid.c
#include <sys/syscall.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

pid_t getpid(void)
{
    return 1;
}

# gcc -fPIC -shared -o getpid.so getpid.c
# export LD_PRELOAD=./getpid.so
# ./soft
pid : 1
```

Figure 18: Custom getpid() library

As we can see, the program launches our library. To avoid a program alteration, we can display the information and after launch the "real" function:

```
# cat getpidV2.c
#include <sys/syscall.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <dlfcn.h>

pid_t getpid(void)
{
    printf("Hello World!\n");
    static void * (*func)();
    func = (void *(*)(void)) dlsym(RTLD_NEXT, "getpid");
    return(func());
}
```

Figure 19: Custom getpid() library V2

This feature can be used to print arguments passed to a specific function and execute it after printing. It is useful to validate if those arguments are coherent during a fuzzing test.

This method is a static method, but we can use a dynamic method to modify a function on the fly.

On Linux, we can use `ptrace()` or `Pin` to attach code to a running process and inject/replace code.

On Windows, we can use DLL injection by forcing a running process to load dynamic-link libraries. .

The result of a dynamic injection is the same as a static injection as seen before.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 62 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

1.7.3 Debugging

As described on Wikipedia¹²: **Debugging** is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware, thus making it behave as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another. The debugging involves numerous aspects, including: interactive debugging, control flow, integration testing, log files, monitoring (application, system), memory dumps, profiling, Statistical Process Control, and special design tactics to improve detection while simplifying changes.

During a fuzzing test, we may consider two types of debugging: live debugging (dynamic debugging during the execution of the program) and post-mortem debugging (after the application has crashed).

1.7.3.1 Live debugging

A specific tool called debugger is used to debug an application. This tool monitors the execution of a program, stops it, re-starts it, sets breakpoints and changes values in memory.

A breakpoint is an intentional pause in a program. During this pause, we can inspect the environment as registers or memory. After the pause, we can execute the program step by step (in fact ASM instruction by ASM instruction). During a fuzzing test, it is necessary to control if all data of a specific function is correct and does not involve a buffer overflow for example.

To debug, we can use debuggers as OllyDBG, Immunity Debugger, IDA Pro for Windows and GDB for Linux. In the examples bellow, we will use Immunity Debugger, which is a free tool.

To illustrate how a debugger works, we simply consider this code:

```
#include <stdio.h>
#include <strings.h>

int dummy(char *string)
{
    char buffer[10];
    strcpy(buffer, string);
    return 0;
}

int main(int argc, char **argv)
{
    int a;
    scanf("%d",&a);
    dummy(argv[1]);
    return 0;
}
```

Figure 20: Example of buffer overflow

If the user put more than 10 characters on the first option, the program will crash due to the `strcpy()`. We can use the debugger to identify this bug. The first step is to identify the function `dummy()` in ASM:

```
.text:004013C0 sub_4013C0          proc near                               ; CODE XREF:
sub_4013DF+2Dp
.text:004013C0
.text:004013C0 var_28             = dword ptr -28h
.text:004013C0 var_24             = dword ptr -24h
.text:004013C0 var_12             = dword ptr -12h
.text:004013C0 arg_0              = dword ptr  8
.text:004013C0
```

¹² <http://en.wikipedia.org/wiki/Debugging>

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 63 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

.text:004013C0	push	ebp
.text:004013C1	mov	ebp, esp
.text:004013C3	sub	esp, 28h ; char *
.text:004013C6	mov	eax, [ebp+arg_0]
.text:004013C9	mov	[esp+28h+var_24], eax
.text:004013CD	lea	eax, [ebp+var_12]
.text:004013D0	mov	[esp+28h+var_28], eax
.text:004013D3	call	strcpy
.text:004013D8	mov	eax, 0
.text:004013DD	leave	
.text:004013DE	retn	
.text:004013DE sub_4013C0	endp	

Figure 21: dummy () function disassembled

So we can put a breakpoint at the address: 0x004013C0. On the debugger, the user simply presses F2 on the good memory address and it becomes blue:

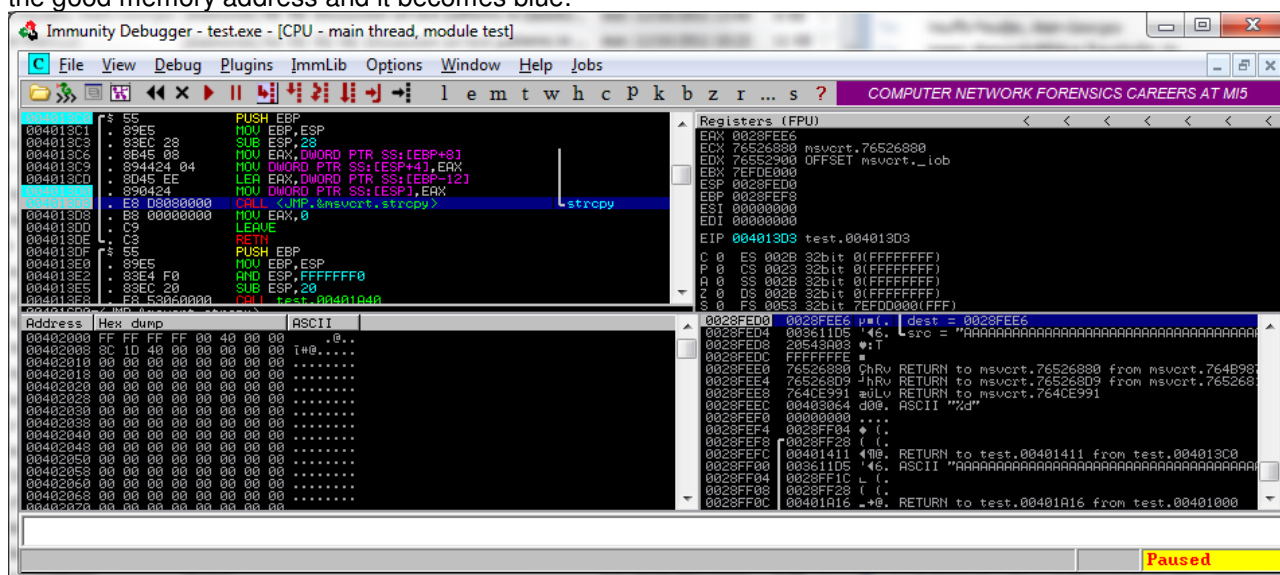


Figure 22: Immunity debugger at dummy () function

At this moment, we can use F7 to execute the program instruction by instruction. We can see the stack on bottom right frame. It is important to examine it; the stack contains the return address of the dummy function which corresponds to the address to jump to when the function finishes (in this case: RETURN to test.00401411 from test.004013C0). After the strcpy() function, we can see on the stack that the return address of this function is overwritten by AAAA (0x41414141).

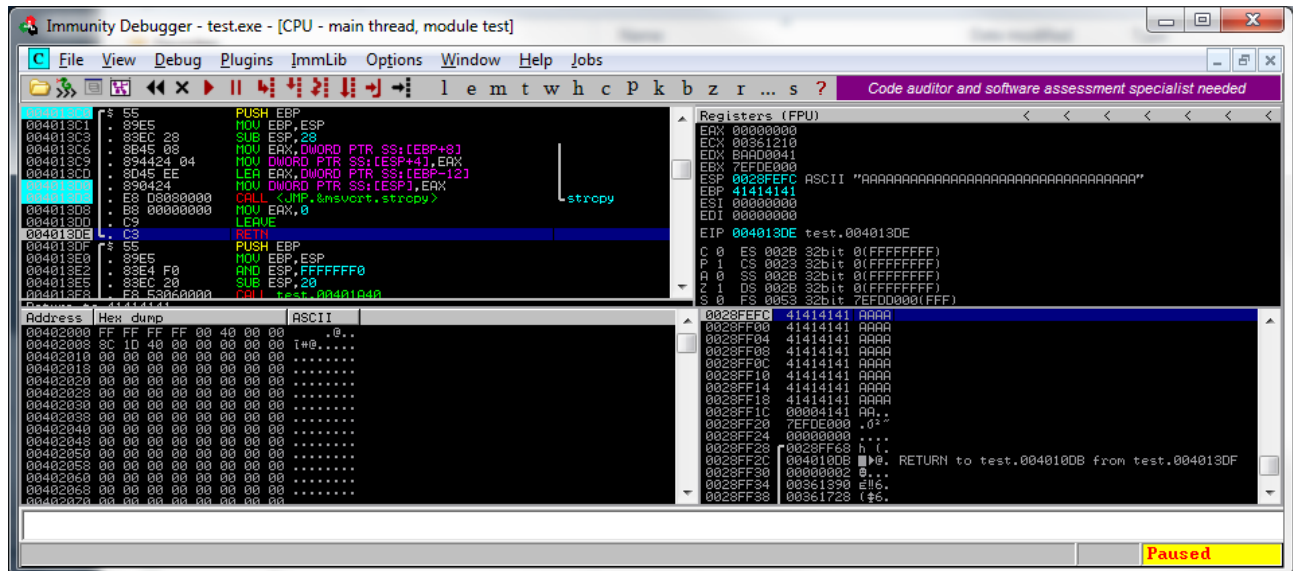


Figure 23: Immunity debugger show how the application crashed

The program cannot jump to the address 0x41414141 so the program crashes.

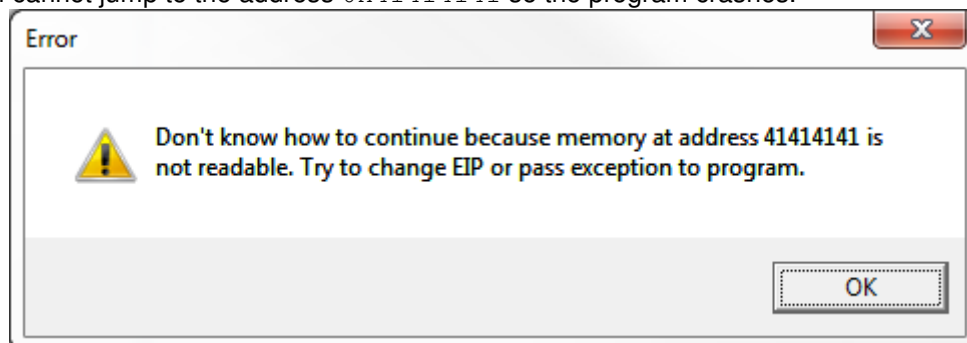


Figure 24: Application crashed

This example shows how it is important to use a debugger during a fuzzing test to clearly identify the bug's provenance.

1.7.3.2 Post-mortem debugging

Unlike the live debugging, the post-mortem debugging involves the analysis of the application after the crash and not during the execution.

To understand the post-mortem debugging process, we will use the same source code used in the previous section. Now we will run GDB tool on a Linux OS and perform the post-mortem debugging.

The first step is to generate a core dump:

```
user@itrust-dev:~$ ./a AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
1
Segmentation fault (core dumped)
user@itrust-dev:~$ file core
core: ELF 32-bit LSB core file Intel 80386, version 1 (SYSV), SVR4-style
```

Figure 25: Core dump generation

The second step is to analyse the core dump with GDB:

```
user@itrust-dev:~$ gdb ./a core
GNU gdb 6.8-debian
```

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 65 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

```
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
```

```
warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/tls/i686/cmov/libc.so.6...done.
Loaded symbols for /lib/tls/i686/cmov/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Reading symbols from /lib/libgcc_s.so.1...done.
Loaded symbols for /lib/libgcc_s.so.1
Core was generated by `./a AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 6, Aborted.
[New process 24129]
#0  0xb7f8b410 in __kernel_vsyscall ()
(gdb) bt
#0  0xb7f8b410 in __kernel_vsyscall ()
#1  0xb7e57085 in raise () from /lib/tls/i686/cmov/libc.so.6
#2  0xb7e58a01 in abort () from /lib/tls/i686/cmov/libc.so.6
#3  0xb7e8fb7c in ?? () from /lib/tls/i686/cmov/libc.so.6
#4  0xb7f19138 in __fortify_fail () from /lib/tls/i686/cmov/libc.so.6
#5  0xb7f190f0 in __stack_chk_fail () from /lib/tls/i686/cmov/libc.so.6
#6  0x08048443 in dummy ()
#7  0x41414141 in ?? ()
```

Figure 26: Core dump analysis

As we can see the last called function in the program is `dummy()` so we can conclude that this function contains the bug that has crashed the program.

1.8 SUMMARY

This section described the contribution of DIAMONDS partners in the context of model based monitoring and inspection. The concepts presented are complementary and present an interesting background to build innovative model-based monitoring techniques.

Unlike active testing, passive monitoring does not inject traffic in the network or modify the traffic that is being transmitted in the network. Nevertheless, active testing can be very crucial to stimulate critical systems and detect vulnerabilities and security flaws. Concepts of active testing will be presented in the next section.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 66 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

2. ACTIVE TESTING CONCEPTS

2.1 INTRODUCTION

Chapter 2 discusses methods for active testing with the objective to generate security relevant test case directly from available models. Approaches that are discussed in this chapter include conformance testing, model extraction, fuzzing, and mutation-based test case generation. In conformance testing a model, a certain safety property, and a test purpose have to be available. The model captures the behavior of the system usually in terms of finite automata or other directed graph structures. The test case generation method uses the test purpose and the properties to extract the test cases from the model. The underlying methods in case of security testing are the same than the methods used for extracting functional test cases.

Extracting models from a system under test (SUT) is a good idea because it prevents from writing models, which requires additional effort. The described method makes use of static and dynamic extraction methods. Static methods for example allows for giving back structural information of the SUT, e.g., control flow graphs. Dynamic analysis allows for deriving use patterns or the frequency of execution of a certain part of the SUT. The proposed method uses the obtained model and generates abstract test cases. The underlying idea here is to compare the model with vulnerability patterns. From the abstract test cases concrete, i.e., executable, test cases can be obtained using fuzzing.

When dealing with model-based testing using model mutations strategies for test case generation become important. This is especially true when the aim is to produce test cases for practical applications where the models usually are larger. Thus test case generation quickly becomes unfeasible and there is a need for improving the test case generation methodology. Therefore, one part of this chapter describes different test case generation strategies in the context of mutation-based testing and compares them with respect to running time and the quality of the generated test suites in terms of their capability to detect faults. Knowing optimal strategy is important also within the DIAMONDS project to make test case generation methods applicable in a practical setting.

Another important part of active testing is the question of the modeling language used. UMLSec is an extension of UML developed for security applications. Relying on standards like UML is a good idea to increase the applicability of approaches. In this chapter the use of UMLSec to specify CORAS risk models (and in particular the conversion of such risk models to UMLSec) is described. Another method introduced in this chapter also makes use of UML. The idea behind is to apply Fuzzing to UML sequence diagrams. In this method basically mutation operators like moving messages or negating constraints and conditions are used to find test cases that potentially break code.

The methods described in this chapter provide a valuable basis for further research and applications in the context of active security testing.

2.2 MODEL-BASED SECURITY TESTING FROM TEST PURPOSES

Model-based security testing from test purposes is an extension of model-based testing (MBT) which targets the conformance testing of security functions w.r.t the specifications but also vulnerability testing. Figure 27 illustrates the global process of model-based security testing with four steps.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 67 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

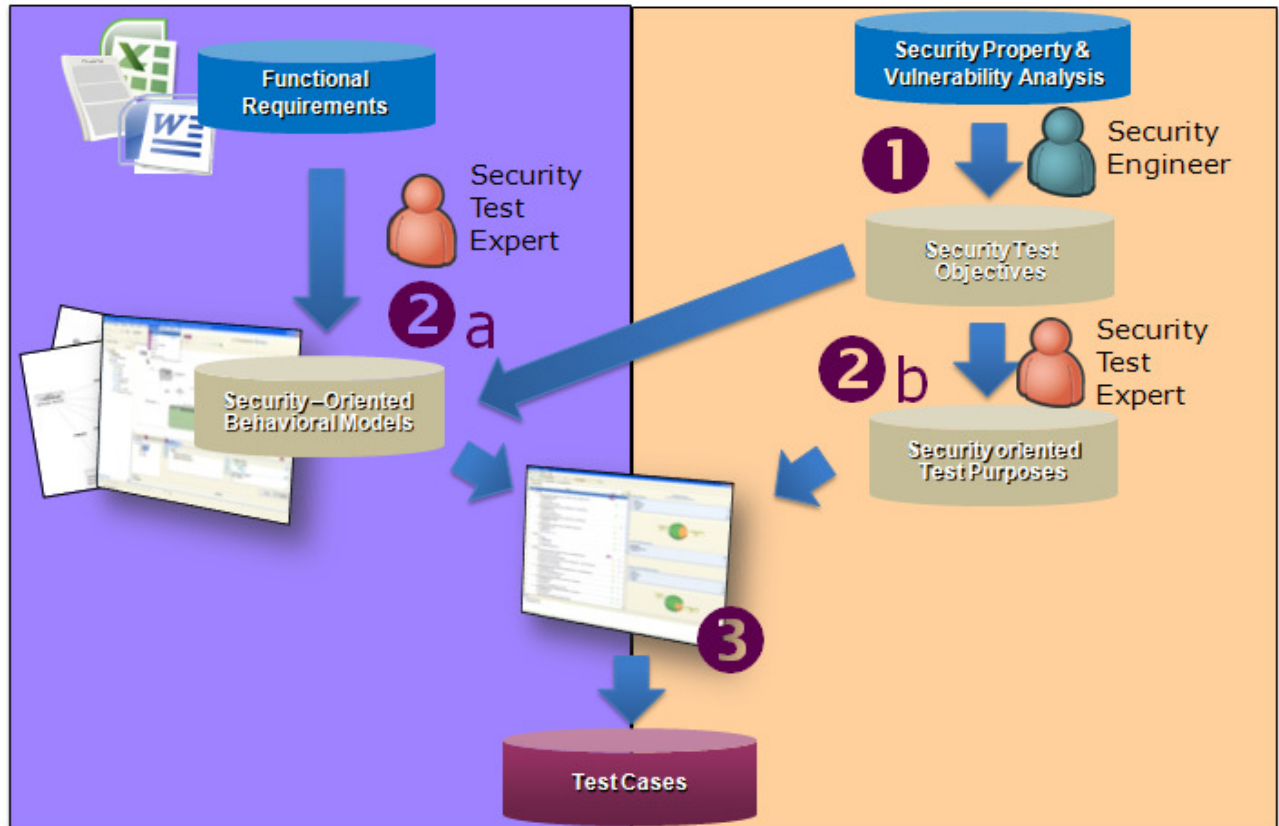


Figure 27: Process of model-based security testing with test purposes

- **Step 1 – Defining Security Test Objectives.** From the security property and vulnerability analysis, a Security Engineer defines the security test objectives. These expose in a detailed (but informal) the statements of test objectives for security testing, and possibly way of testing (how to test). These security test objectives define the testing strategy and impact both the modelling activities and the driving of automated security test generation.
- **Step 2 – Modelling.** In this phase, there are two main modelling activities that are delivered by the Security Test Expert in an incremental way:
 - **Step 2a - Behavioural modelling** of the System Under Test (SUT). This model focuses on security features and test vulnerabilities. In fact, the behavioural model is restricted on the features relevant to the security test objectives. The model formalizes the relevant point of control and observation, and the expected behaviours of the SUT. This behavioural modelling activity differs from classical functional model-based testing, because the security-oriented behavioural model integrates non-nominal stimuli (stimuli that are not part of the specifications) but which correspond to possible actions from an attacker. Also, the observation points may be specifics to reinforce the capability of establishing an accurate verdict the generated security tests. The security-oriented behavioural model is build on the basis of the SUT functional requirements and the security test objectives.
 - **Step 2b – Security Test Purpose definition.** Security Test Purposes are a formal definition of the security test objective. They allow explaining the objectives with dedicated test purpose language. So, this formalization is able to express meta-scenario in terms of state and actions in regards of elements of the model. Each security test purpose is unfolded in several (or multiple) tests related to a particular test objective. Figure 28 provides an example of Security Test Purpose in a smartcard application domain.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 68 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

- **Step 3 – Automated test generation.** From the security-oriented behavioural model and from security test purposes, security test cases and test scripts are generated. At this stage, the same process with functional MBT applies (see [67] Utting 2007): generated tests are published in a test repository, and test automation required a development of a so-called adaptation layer to implement the action words defined in the behavioural model.

Within the Diamonds project, this process is supported by the Smartesting approach for Security testing. The Smartesting prototype, developed within the Diamonds projects, takes as an input security-oriented behavioural model using UML/OCL notation and security-test purposes based on a dedicated language. This prototype and associated concepts is presented in detail in Diamonds deliverable D2.WP3.Initial_design_of_security_testing_tools.

Figure 28 present an example of a test purpose in the Smartesting and it is unfolding after test generation in four different test scenarios.

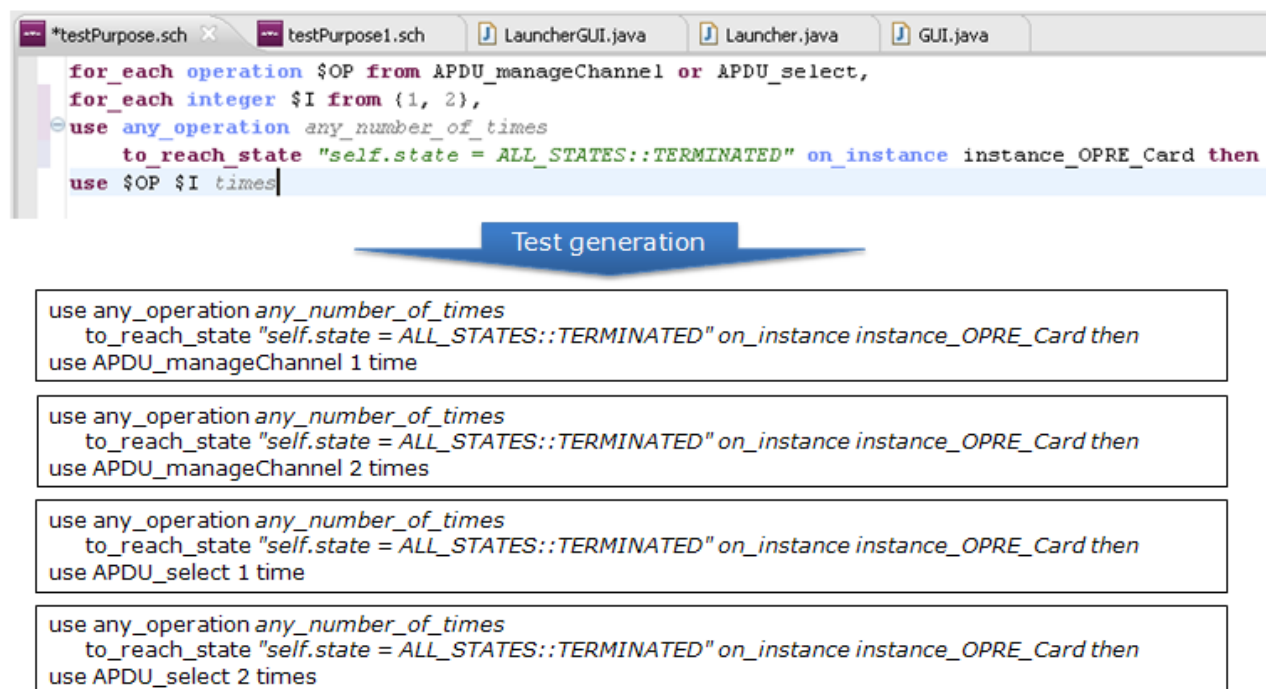


Figure 28: Test purpose example and unfolding

To summarize, model-based security test generation is based on security-oriented behavioural model and security test purposes. It targets the testing of security functions in a SUT and vulnerabilities to be tested on the basis of the Security Test Objectives. In this approach, the Security Test Expert is in charge of the development and the maintenance of the models and the test purposes, and he/she drives the test generation using a test generation tool.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 69 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

2.3 INTEGRATION OF BASIC SECURITY RULES INVOLVING ATOMIC ACTIONS

Security and reliability are important issues in designing and building systems with time constraints because any security failure can be risky for users, their business and/or their environment. Currently, software engineers developing systems, with time constraints, are not only confronted to their functional requirements but they also have to manage other aspects concerning security issues. We mean by 'functional requirements' the services that a system offers to end users. Whereas, security rules denote the properties (restrictions) that a system has to fulfill to be always in a safe state, or also to guarantee good quality of services it provides. For instance, a file system may have to specify the prohibition for a user to access to a specific document if he/she is not authenticated or if his/her session of 10 minutes has been expired. More generally, a security rule expresses the obligation, permission or interdiction to perform an action under given conditions called context.

Complex systems are often incrementally designed. They are evolutionary where new requirements may appear during its life cycle and then have to be integrated to its initial specification. In this paper, we deal with a particular kind of requirements denoting security properties. Basically, we provide a formal approach to integrate elaborated security rules involving time constraints into a system specification based on communicating extended timed automata called TEFSM specification¹³. The resulting model can be used to generate code or the automatically derive security-oriented test cases.

We use the Nomad¹⁴ formal language to specify without any ambiguity the set of security properties that the system has to respect. The choice of this language is mainly motivated by the Nomad features that provide a way to describe permissions, prohibitions and obligations related to non-atomic actions within elaborated contexts that takes into account time constraints. By combining deontic and temporal logics, Nomad allows to describe conditional privileges and obligations with deadlines, thanks to the time concept it supports. Finally, it can also formally analyze how privileges on non-atomic actions can be decomposed into more basic privileges on elementary actions.

We describe here the integration of basic security rules of the form $R(\text{start}(A)|O^{[-d]} \text{done}(B))$ where $R \in \{F; O; P\}$, A and B denote atomic actions, and ($d > 0$). Since we deal with a timed context, we need to define a global clock gck to manage the temporal aspect of the rules.

2.3.1 Prohibition integration: $F(\text{start}(A)|O^{[-d]} \text{done}(B))$

The key idea of integrating such a prohibition rule in a TEFSM model is to check the rule context before performing the prohibited action. If this context is verified, the prohibited action A must be skipped. Otherwise, if the context is not valid, the action is performed without any rule violation. To achieve this goal, a table *Prohib* has to be built to store all the instants (or moments) where it is prohibited to trigger a given transition since it contains the forbidden action A . These instants are denoted by a predicate on clock gck which is a global clock for the system launched in its initial state. For instance, a predicate $((gck < 10) \vee (gck = 15))$ means that the execution of a transition tr is prohibited till the tenth unit of time and also at the fifteenth since it contains a prohibited action. We also define the function $\text{val}(gck)$ that provides the clock value gck at a specific moment. Table *Prohib*(tr) is updated as follows (tr denotes a transition where action A appears):

- After each occurrence of B in the TEFSM transitions, the value of *Prohib*(tr) is updated by adding a predicate on the instant(s) when it is prohibited to trigger tr since it contains action A . The new value of *Prohib*(tr) is defined by:

$$\text{Prohib}(tr) = \begin{cases} \text{Prohib}(tr) \vee (gck < \text{val}(gck) + d) & \text{for } F(\text{start}(A)|O^{[-d]} \text{done}(B)) \\ \text{Prohib}(tr) \vee (gck = \text{val}(gck) + d) & \text{for } F(\text{start}(A)|O^d \text{done}(B)) \end{cases}$$

¹³ M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis, "The IF Toolset," in SFM, 2004, pp. 237–267.

¹⁴ F. Cuppens, N. Cuppens-Boulahia, and T. Sans, "Nomad: A Security Model with Non Atomic Actions and Deadlines," in CSFW, 2005, pp. 186–196.



Concepts for Model-Based Security Testing

Deliverable ID: D2.WP2

Page : 70 of 134

Version: 1.0

Date : 26.09.2011

Status : Final

Confid : Public

- Before triggering the prohibited transition tr , we check whether the value $val(gck)$ satisfies $(Prohib(tr))$ to deduce if tr can be fired or not.

Notice that $Prohib(tr)$ can be updated according to all the security rules that deny the execution of any action A that may be executed in tr . Depending on each context, the update of $Prohib(tr)$ is performed as mentioned above.

The prohibition integration methodology performs a primary processing of the initial TEFSM specification, so that no transition would contain the prohibited action A after action B . Figure 29 illustrates this primary phase by decomposing the transitions where prohibited action A appears after action B .

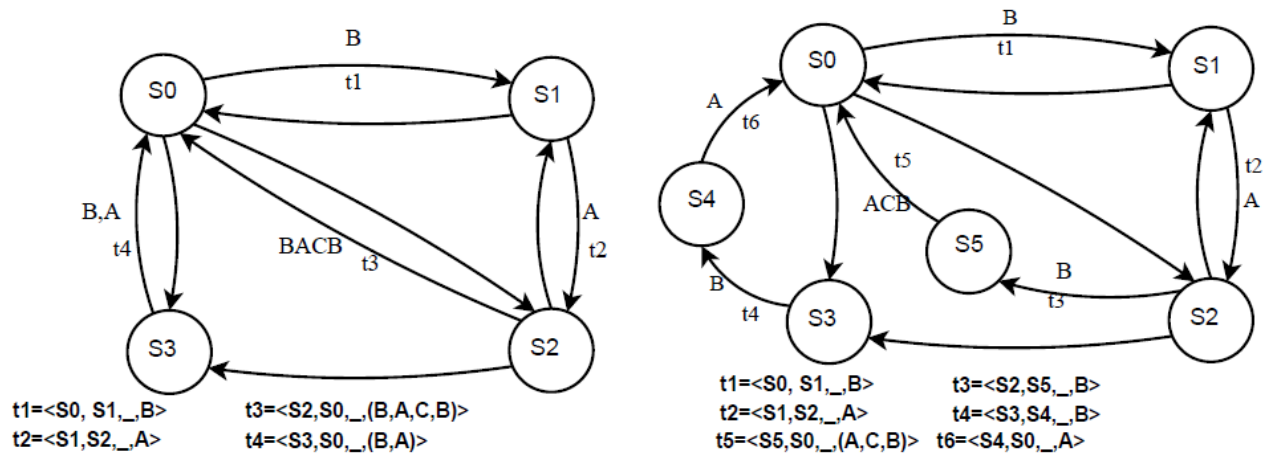


Figure 29: Transition decomposition

Transition ($t3 = \langle S2; S0; -; (B;A;C;B) \rangle$) for instance has been split into two transitions ($t3 = \langle S2; S5; ;B \rangle$) and ($t5 = \langle S5; S0; -; (A;C;B) \rangle$) by introducing the new state $S5$. In the resulting specification, we want to integrate the rule $F(start(A) / O^{<d} done(B))$ which stipulates that it is forbidden to perform action A within d units of time of B being performed. The application of the above mentioned algorithm produces the secure system depicted in Figure 30.

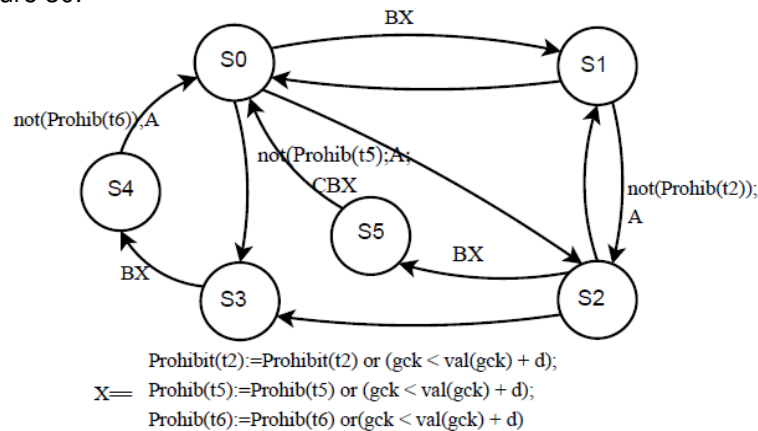


Figure 30: Prohibition rule integration: $F(start(A) / O^{<d} done(B))$

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 71 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

2.3.2 Permission integration: $P(\text{start}(A) | O^{[<]-d} \text{done}(B))$

The algorithm to integrate a permission rule is very similar to the above mentioned algorithm. A variable $\text{Permis}(tr)$ is defined for each transition that contains an action A appearing in a rule of the form $P(\text{start}(A) | O^{[<]-d} \text{done}(B))$. $\text{Permis}(tr)$ stores all the instants where it is permitted to trigger tr ; it is updated exactly as $\text{Prohib}(tr)$ each time action B occurs. Finally, we have to check that predicate $\text{Permis}(tr)$ is satisfied before triggering transition tr .

2.3.3 Obligation integration

Since different obligation rules related to action A may be defined, we have to take into account the possible dependencies that may exist between them. In fact let us consider for instance rules $O(\text{start}(A) | O^{-5} \text{done}(B))$ and $O(\text{start}(A) | O^{<-10} \text{done}(C))$ where action C is executed 3 units of times (less than 5 units of times) after the execution of B . The execution of action A five units of time after the execution of B satisfies both rules at the same time. The idea behind this simple example is to check if it necessary to execute the mandatory rule for each execution of the context action. To integrate an obligation security rule in the TEFSM based system specification, we rely on a new process RHP which ensures the execution of the mandatory action. If the related mandatory action is not executed by the initial specification, process RHP then has the task to execute it itself. The integration methodology follows these steps for a rule in the form of $O(\text{start}(A) | O^{<-d} \text{done}(B))$ where $d > 0$:

- A boolean variable wait_A is defined. It checks whether we are waiting for the execution of an instance of action A or not. This variable is set to true at the execution of each action B for which an obligation rule $O(\text{start}(A) | O^{[<]-d} \text{done}(B))$, and set to false when action A is executed.
- The definition of a new process that can be created (forked) n times by the initial functional specification, where n is the maximum number of simultaneous executions of action B in the initial TEFSM specification. This new process has two parameters. The first parameter, equal to $(\text{val}(gck) + d)$ (resp. $(\text{val}(gck) + (d - 1))$), states the instant when (resp. before which) action A should be executed if we deal with the O^{-d} timed operator (resp. $O^{<-d}$). The second parameter exactTime is a boolean variable which determines whether action A must be executed at $(\text{val}(gck) + d)$ (for a rule of the form $O(\text{start}(A) | O^{-d} \text{done}(B))$) units of time or before this moment (for a rule of the form $O(\text{start}(A) | O^{<-d} \text{done}(B))$). The process has to wait until the execution deadline of action A is reached. Also, we give it a lower priority with respect to the main process in order to let this last executes its actions at first. To do that, we declare the transitions of this new process as *delayable*. A *delayable* transition allows time progress unless time progress disables it, in that case it is taken. The deadline and the actions to perform ($\text{deadline}; \text{Action}$) depends on the type of rule and whether we are waiting for an execution of action A :

$$(\text{deadline}; \text{Action}) = \begin{cases} (gck = \text{val}(gck) + d, A; (\text{wait}_A := \text{false}); \text{stop}) \text{ for } O(\text{start}(A) | O^{-d} \text{done}(B)) \\ ((gck = \text{val}(gck) + (d-1)) \vee \neg \text{wait}_A; \text{if } \text{wait}_A \text{ then } (A; \text{stop}); \\ \text{for } O(\text{start}(A) | O^{<-d} \text{done}(B))) \end{cases}$$

In Figure 31, it is presented the integration of an obligation rule within the initial system depicted in Figure 29. In this functional system, we can find several occurrences of the atomic action B .

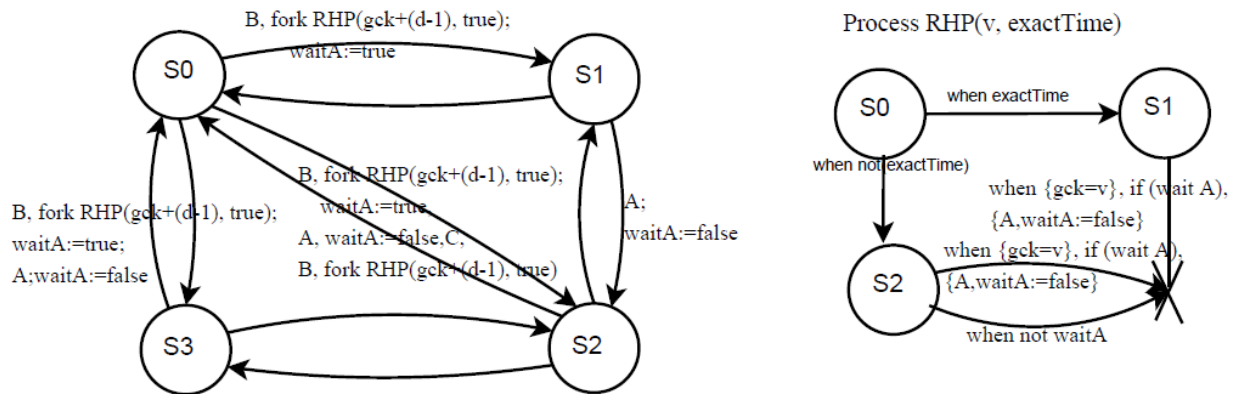


Figure 31: Obligation Rule Integration

2.4 COMBINING MODEL-RECOVERY AND EVOLUTIONARY FUZZING

The aim of security testing is to find vulnerabilities in the SUT that could be exploited to subvert the security of the SUT. According to OWASP, “A *vulnerability* is a hole or a weakness in the application, which can be a design flaw or an implementation bug, that allows an attacker to cause harm to the stakeholders of an application.”

2.4.1 The General Approach

The objective of this work is to use testing techniques to exhibit security vulnerabilities on a System Under Test (SUT) by combining three main steps:

- 1) model recovery, which consists in building (or retrieving) a (partial and abstract) behavioral model of the SUT;
- 2) model-based vulnerability detection, to provide some abstract test cases as model-level execution sequences that may activate some vulnerabilities;
- 3) evolutionary fuzzing, whose goal is to concretize those abstract test sequences following a mutation-based input generation guided by a fitness function.

A high-level view of this general approach is illustrated in Figure 32:

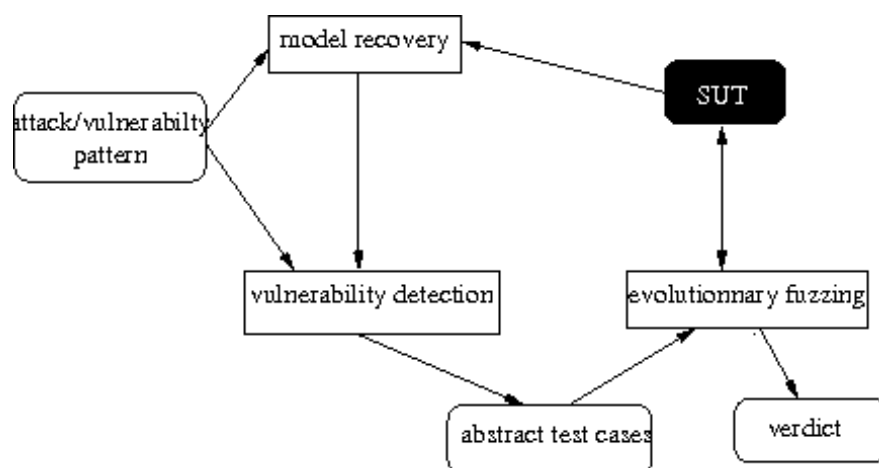


Figure 32: A general view of the proposed approach

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 73 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

a) Model recovery

The goal of this step is to build a model of the SUT able to (partially) describe its behavior at some abstract level. This model recovery can be performed following several techniques, according to the initial knowledge available on the SUT (e.g., source/binary code, input formats, main use cases, etc.). Depending on this knowledge this model may be obtained using a dynamic analysis (by means of well-chosen executions of the SUT), or by a static code analysis (based on reverse engineering techniques).

In the former case the model obtained express a set of (parameterized) interaction sequences/trees between the SUT and its external environment. In the latter case, this model reflects the code structure of a SUT, for instance as a control-flow graph.

b) Vulnerability detection

Once a model has been obtained, the next step consists in looking for potential vulnerabilities. This search can be guided by some attack or vulnerability patterns, defined at the model level. Again, the technique to use may depend on the nature of the model (model-checking, syntactic recognition, static analysis, etc.).

The result is a set of abstract execution sequences (defined on the model) that may lead to potential vulnerabilities (i.e., matching some vulnerability patterns). Note that these sequences may refer to a possibly unsound and incomplete behavioral model of the SUT.

c) Evolutionary Fuzzing

This next step aims at turning the abstract test sequences into concrete test cases able to activate effective vulnerabilities on the SUT. It is based on a so-called evolutionary fuzzing, which consists in computing successive improvements of an initial random set of (concrete) SUT input sequences by mutations and crossovers. Each input sequence leads to an execution sequence which is scored by a fitness function. This function measures:

- the deviation of the concrete execution sequence from the expected abstract one (which is supposed to lead to a vulnerability);
- how able is this concrete sequence to activate this vulnerability.

This process stops either when the vulnerability is revealed (then the test is successful), or when a maximal number of executions has been reached (then the test fails). Let note that these test executions may also be used to refine the initial model (and hence the abstract test cases).

In the following sections we describe two instances of this general approach corresponding to two specific contexts:

- a black-box instance, able to detect design-level vulnerabilities;
- a white-box instance, able to detect code-level vulnerabilities.

A long-term perspective could be to combine these two approaches on a same case-study to address a larger range of security vulnerabilities.

2.4.2 Smart Black-Box Fuzzing

In this section we instantiate the approach considered in the case where we do not have access to the source code of the SUT, so all the tests are done in black box mode. The result is shown in Figure 33.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 74 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

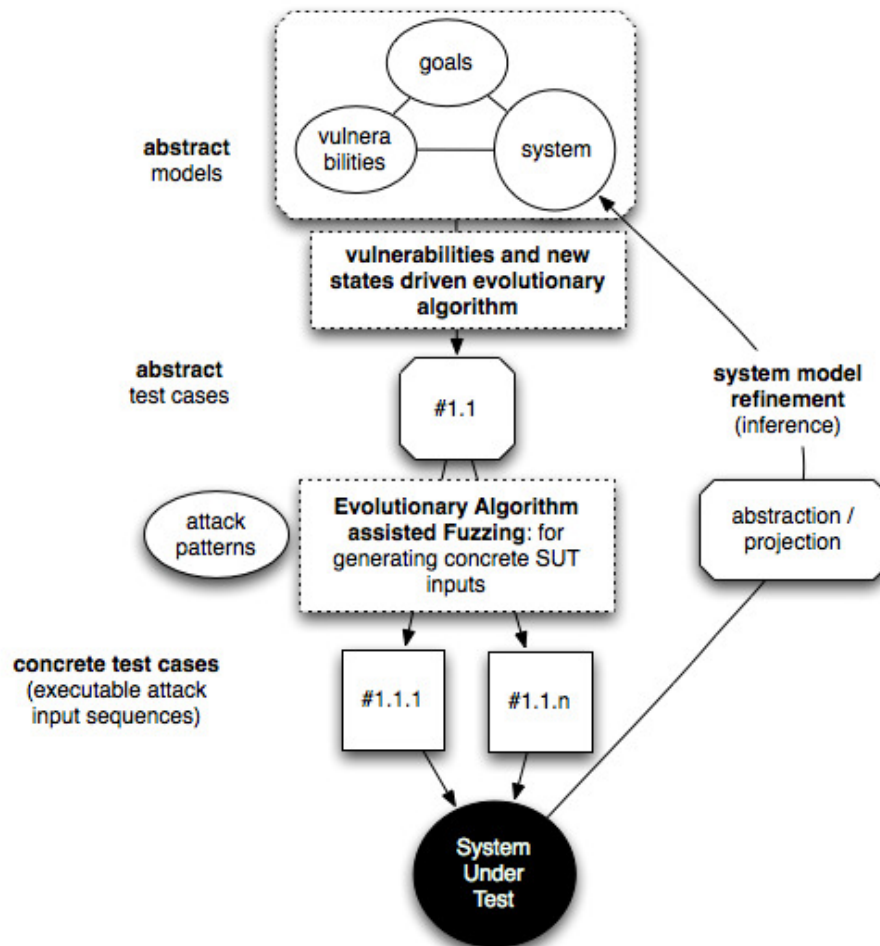


Figure 33: Our proposed approach on combining model inference and fuzzing for automated vulnerability search on a black-box SUT

a) Combining model-inference, evolutionary algorithm and fuzzing: approach overview

a.0) Assumptions: it is assumed that the SUT input and output symbols and their parameters (types and structures) are known. Also that the security goals/objectives – that are security properties we want to verify on certain set of SUT model parameters - are modeled. Note that such properties would usually be simple (e.g. allowing/denying access to a resource, absence of crash, etc.). They could be expressed either as state properties, or more generally as trace properties described by temporal logic.

a.1) Producing abstract vulnerabilities models that might exist in the SUT. A vulnerability model contains at least one state that violates at least one security goal, or in the more complex case, a trace that violates the goal.

a.2) Vulnerability driven evolutionary exploration: the two main goals are to produce an abstract model of the SUT and to guide the test execution towards vulnerable states. For that, input sequences will be submitted to the SUT in order to discover new states, but also once a condition (e.g. discovery of a state similar enough to one present within a vulnerability model, number of inference steps, number of discovered abstract states..) has been fulfilled. Then several abstract test cases will be created to put the execution flow towards states that have the strongest similarity with states present within at least one vulnerability model.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 75 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Model recovery could thus use similarity metrics between states. Since an evolutionary algorithm will be used, the fitness function will take those variables into account for guiding the exploration wrt to the two stated goals.

a.3) From abstract to concrete test cases – evolutionary fuzzing and attack patterns: once an abstract vulnerability-pattern has been found (modulo a similarity metrics), the goal is to generate concrete input sequences for verifying whether it could correspond to a vulnerability within the implementation. For that purpose, evolutionary fuzzing and attack patterns (regarding the specific abstract vulnerability being tested) will be combined for enhancing the concrete value space.

a.4) Abstracting the SUT outputs: each time a concrete test is executed on the SUT, a potentially new knowledge is obtained. Once abstracted/projected, the SUT outputs will be processed through an adaptation of the [31] inference algorithm, and abstract input scoring will then be performed.

a.5) Iteration: the testing process will go back to 2, if no concrete input sequence exhibiting the potential existence of the vulnerable state was generated.

b) More precise approach

b.1) Abstract vulnerability model generation

Starting from generic parameterized vulnerability models and security objectives, generate abstract vulnerability models. A vulnerability model contains at least one state that violates at least one security objective on a set of SUT parameters.

In [62] interesting results were obtained by testing a given implementation A wrt to vulnerabilities We consider doing something similar in that black-box fashion.

b.2) Vulnerability driven evolutionary exploration

An evolutionary algorithm will be used to guide the exploration process. Here is the pseudo-code of such an algorithm:

```

-----
INITIAL_POPULATION = accepted SUT inputs inferred so far (might be empty)
REPEAT
• SELECT parents
• RECOMBINE parents
• MUTATE offsprings
• EVALUATE candidates with FITNESS function
• SELECT the fittest candidates for next population
UNTIL TERMINATION_CONDITION is met

Return SOLUTIONS, if found
-----

```

At that level, mutation consists either in adding an input symbol to a given input sequence or mutating the final input symbol (in particular its parameters).

The fitness function will assign a higher score to input sequences that led to discovering new states in the SUT and to ones that led the execution towards states that are “similar” to ones within at least one vulnerability model. One important thing to notice is that scoring will be possible after step 4 (output abstraction) and the inference process.

Similarity metrics will be computed regarding partial matches such as input and output symbols and or parameters. Methods such as state distinguishability regarding a set of input sequences as stated in [27] will be investigated.

b.3) from abstract to concrete test cases: evolutionary fuzzing and attack patterns

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 76 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

During that step, concrete executable tests are generated from abstract tests. For that purpose, an evolutionary algorithm (EA) will also be used, in which the population is the concrete SUT inputs and the genes are the instantiated input symbols and parameter values.

Attack patterns depend on the given vulnerability being tested. They will influence how the inputs will be fuzzed (recombined and mutated). Input parameters will be fuzzed wrt their type values (e.g.: 2^i-1 V i in $[0..n-1]$ if we assume that the integer is unsigned on n bits) but also their representation and the messages structure (eg: voluntarily invalid message format within a XML message, incoherent headers...).

The recombination step is performed using several crossover methods (cross sessions, cross transitions, cross of parameters within the same transition). Note that the input parameters in the model are typed (e.g.: integer, string, xml structure...). Thus crossover will be performed wrt. those types.

The fitness function will take as arguments the fitness score assigned and some kind of randomness, to avoid getting a flat value space.

b.4) abstracting the SUT outputs and inferring

Concrete output values will be abstracted. Inference then takes place in order to determine if the reached abstract state is a new one or not. To decide this, discriminating abstract input sequences are sent to the SUT. Thus until it is determined if the reached abstract state is the same as another one or not, such input sequences will have a temporary increased scoring.

b.5) Eventual iteration

The `TERMINATION_CONDITION` is evaluated to true when a security goal has been violated. It could be any of the following stopping criteria: availability (e.g.: in case of a SUT crash or too long delays for processing an input) or reaching a vulnerable state (within a vulnerability model).

Finally, if the algorithm stops we have at least one concrete input sequence for the SUT leading to a state that violates a security goal.

c) Remark regarding model Inference

During the process (and especially at steps 2 and 5), the SUT will be actively inferred using a modified version of the [31] algorithm.

Model inference consists on automatically building a model of the SUT by observing and abstracting behaviors in terms of inputs/outputs. In the hacking terminology some would refer to this as an automated reverse engineering method.

Two categories of model inference do exist: active learning and passive learning. Passive learning checks that some invariant properties are not violated by observing captured traces. Active learning is free to interact with the system. In situation where this assumption holds, we chose to focus on active learning since it will converge faster than passive learning.

Angluin's L^* algorithm [1] is a famous example of an active learning algorithm. Some algorithms might however be more suitable for inferring transition with parameters [47][31] (especially in terms of number of inferred states). We will use that last one which better corresponds to security concerns.

2.4.3 Smart White-Box Fuzzing

In this section we instantiate the approach in the case where we can access the code of the SUT, so that at least its control flow can be retrieved by mostly by static analysis.

a) Code analysis assisted smart fuzzing

The focus of this work is on *implementation bug*. Fuzzing has been proved to be very effective in finding implementation bugs. However, given the complexity of modern applications, using the fuzzing in its traditional form i.e. blind fuzzing, may not be very effective always. By considering the factors such as how an application process the information/data, how does it interact with its environment etc. in the process of fuzzing may help to fuzz in a better way and thereby making the fuzzing more effective i.e. *smart fuzzing*. One of such factors is how does the application process the data, and specially the input that is *tainted*. In

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 77 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

order to observe this behavior, we need to consider a suitable model of the application, obtained from its source or binary code. By analyzing this model, we can establish two things:

1. There exist a bug (a pattern that exhibits weakness)- target or sink
2. Tainted input (source) can reach this target.

In essence, by analyzing the code, we can show, conservatively, that there exists a bug and a path from source to sink which could be used to exploit the vulnerability. However, to eliminate false positives, such a static approach needs to be completed by information available only at runtime. A dynamic analysis of the application involves the generation of concrete inputs, expected to trigger vulnerabilities. In the following sections, we elaborate our approach.

b) Overview of the Approach

The ultimate aim of the approach is to generate inputs that can trigger the vulnerability residing inside the SUT. The approach we propose has three distinct features.

1. We are focusing on *buffer overflow type vulnerability*, which are still ranked very high among other serious vulnerabilities [15]. Traditionally, studies which focus on BoF relies on the presence of infamous “strcpy” family of functions as sink. We argue that there may be other functions of similar functionality and therefore, by considering only the known (infamous) functions gives us a false sense of BoF detection. In this study, we devise a method to find similar “interesting functions” by describing the notion of “*interesting loops*” and a method to detect them (in binary code, see below).
2. We analyze the SUT on its compiled executable form which means that our whole analysis is done on the executable of the application. This is challenging, but on the same time provides us the opportunity to analyze the code that is closest to the machine (WYSINWYX [14]) and also widens the scope to analyze COTS software and library functions. The side effect of this analysis is the generation of abstract test cases as *tainted paths* from source to the sink. In our case, source are the functions that accept tainted data and sink are the functions found in item 1 above.
3. Once we get the tainted path, we proceed further by generating inputs those goal is to trigger vulnerabilities. Choosing relevant inputs is crucial since, as while doing an abstract tainted path calculation, we may not be aware of the semantics of some intermediate function that may sanitize the input data before it reaches the target. In this work, we propose to use evolutionary strategies to generate such inputs. Our fitness function is based on the tainted path that is calculated in the above step.

Following, we describe each of these contributions in details.

c) Interesting Functions as targets

From vulnerability detection standpoint, interesting functions are procedures in program that introduce some kind of vulnerability in the program. In the current work, we focus on *string buffer overflow* type of vulnerabilities. More specifically, we are interested in functions that copy strings from one buffer to another. strcpy() family of C/C++ functions is such an example, which are known to the world. Our proposition is that it is the *functionality*, and not the function (name) that makes it vulnerable. There could be several functions that exhibit similar functionality as that of strcpy(). Indeed, several vulnerabilities discovered in a recent past were due to such functions¹⁵. This leads to the following definitions:

Interesting Loop: A loop is interesting if there is memory write within the loop **and** that memory is changing within the loop.

Interesting Function: A function is interesting if it contains an interesting loop.

¹⁵One could mention for instance functions wcsncpy() and SSL_get_shared_ciphers() functions in OpenSSL (CVE-ID:2007-5135)

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 78 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

As mentioned, we analyze the binary of the application and therefore, detecting interesting function is very challenging. At binary level, we do not have a very clear understanding of the underlying structures, its variables, memory access etc. In order to capture the patterns for interesting loops, we define several *dependency relations* and formulate an algorithm to detect them. Once the interesting loops and hence the interesting functions are detected, they become the target for generating the tainted path that is explained in the next section.

d) Tainted Information Flow

The idea of taint-flow analysis is originally based on the dynamic analysis aspect of the SUT (e.g. perl taint mode). Thereafter, it is adopted to static analysis aspect in the form of data-dependency analysis from tainted input to the sink. Depending upon the level of analysis, dependency can be among variables/registers or the memory locations. However, at binary level, performing a fine grained data dependency that involves memory has not been much explored area. In this work, we investigate the use of *value set analysis* (VSA) [11] to perform a taint propagation or taint dependency analysis. The result of such analysis is the set of paths from tainted input to interesting functions. *In essence, this analysis will find the path from tainted input to the interesting function such that the input can influence the execution of the interesting loop.*

e) Input Generation with Evolutionary Fuzzing

Once the tainted paths are determined, we still need to verify the vulnerability. In order to do so, we shift from static analysis to dynamic analysis. Our aim is to generate inputs that trigger the vulnerability by executing the interesting function. We know that in order to execute the function, we need to follow the tainted path or to put in other words, *by following the tainted path, we can reach the interesting function and can trigger the vulnerability.* Therefore, we need to generate inputs that follow a certain path in the SUT. In other words, *search for the inputs such that a certain path is covered.* Looking at this perspective, the problem of generating inputs can be thought of a *search problem*. Evolutionary algorithms (e.g. genetic algorithms) are well known techniques for search based problems. As a consequence, evolutionary algorithms have been used in testing. We investigate the use of evolutionary algorithms to generate such inputs to facilitate fuzzing process. The idea is to generate fuzzed inputs such that they can traverse the tainted path. The fuzzing part can be mapped to genetic operators mutation and cross-over i.e. apply mutation or/and cross-over to fuzz the input *sensibly*. As the consequence of our VSA based static analysis, we know that which part (offsets) of the input file influences the target and we can mutate only these parts such that it contains the symptoms of BoF (i.e. long string) and ability to reach the vulnerable function. The initial ideas are presented in [61].

2.5 MUTATION TESTING

Often implementations that pass functional tests may still be vulnerable to attacks and a number of these vulnerabilities are detected after the release of the product despite of rigorous testing. The aim of security and vulnerability testing is to cover a large number of possible security faults and vulnerabilities. One effective way to address this problem is to adapt mutation analysis so that security policies are defined in terms of specific mutation operators yielding a test selection criteria suited for security and vulnerability testing.

Mutation testing is a software testing method which involves modifying the system code in small changes, yielding a faulty system, a **mutant** of the original one. These changes mimic typical errors that a programmer could have made. The goal is to find weaknesses in a test suite and to develop more effective one.

Mutation analysis is typically applied on program code. In this document, however, we discuss mutation testing from *system model* based testing point of view. This approach mutates the system (or design) model and then compares the mutants with the original model to automatically generate test cases and evaluate coverage.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 79 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

2.5.1 Mutation Analysis Process

Mutation analysis process induces faults in to the system by creating many different versions of the system each containing one fault. The basic idea is then to see if the test cases can detect these faults i.e. the test cases are used to execute these faulty systems in order to distinguish the faulty system from the original system. The main objective is to select such a set of test cases that can be used to detect errors. Faulty systems are **mutants** of the original system and a mutant is said to be **killed** by distinguishing the output of the mutant from that of original system. In other words, a test case that detects a fault injected thru a mutation kills the given mutant and if the test suite kills all the mutants, it potentially detects many unknown faults in the implementation. Mutants that are not killed are said to be **alive** and if the mutant produces the same output as the original system, they are **equivalent** and cannot be killed.

Mutation testing and analysis provides a *testing criterion* rather than a *testing process*. Testing criterion is a set of rules that impose some requirements on the test cases. We use *coverage* as a measure of the extent in which the criterion is satisfied; for example reaching a statement is a requirement for statement coverage while killing mutants is the requirement for mutation. Coverage is measured in terms of the requirements and it is defined to be the percent of requirements that are satisfied. Testing criteria gives requirements of how much testing is actually needed and we use *mutation testing as a way to measure the quality of the test cases*.

Mutants represent likely faults that the programmer could have made by a mistake. Each mutant introduces a very small change, i.e. a fault, to the system compared to the original one. Mutants are limited to simple changes to the system on the basis of **coupling effect** meaning that complex faults are coupled with simpler ones in a way that test cases that detect all the simple faults will also detect the complex faults. Mutated versions are created by applying **mutation operators** that are rules applied to the system to create mutants. These are simple syntactic or semantic transformation rules: typical examples are deleting an assignment expression and replacing or inserting new operators to create syntactically legal expressions are examples of typical mutation operators.

The **mutation score** (or **mutation adequacy**) is calculated once all the test cases have been generated. The mutation score is the ration of dead mutants over total number of non-equivalent mutants:

$$\text{mutation score} = \text{number of killed mutants} / \text{total number of non-equivalent mutants}$$

As with any other testing criteria, the goal here is to raise the mutation score to 100% which is indication that all the mutants have been detected and covered by the test cases. This ratio can be increased by adding more test cases that detect live mutants: live mutants reveal inadequacy in the test suite. Equivalent mutants are not counted in the mutation score. It is worth noting that mutation score gives some indications about the extent of the test suite even if a tester manage not to find defects in the implementation using the set of test cases.

2.5.2 Mutation Operators

A mutation operator is simply a syntactic or semantic transformation rule like replacing operator == with !=. The mutants are generated by applying these mutation operators to the original system. Application of these mutation operators should model potential faults that a programmer could have accidentally make, therefore it is of great importance to identify different kinds of faults. Note that ultimately the number and type of mutation operators influences the number and extent of tests that are produced.

Mutation operators are limited to simple changes to the system on the basis of coupling effect. This says that simple faults are coupled with complex faults in a way that test suite that detects simple faults will also detect complex faults. This was first hypothesized in late 70's and demonstrated theoretically in mid 90's.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 80 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

Mutation operators are independent of the specification notation and typical mutation operators for a textual specification notation are for example

- *Associative shift*: change the association between variables
- *Expression negation*: replace an expression by its negation
- *Logical operator replacement*: replace a logical operator (&&, ||) by another logical operator
- *Missing condition*: delete conditions from conjunctions and disjunctions
- *Operand replacement*: replace an operand i.e. a variable or a constant by another syntactically valid one
- *Relation operator replacement*: replace a relational operator (f.ex. <, ≤, =, ≠, >, ≥) by another relational operator

Typical mutation operators for graphical notation such as state chart based notation are for example

- *Missing state*: delete a state from the state chart
- *Missing transition*: delete a transition from the state chart
- *Initial state exchanging*: exchange the initial state to be another state in the state chart so that the state chart execution starts from a different point
- *Source state exchange*: exchange the source state of a transition by another state in the state chart
- *Target state exchange*: the target state of a transition by another state in the state chart

Mutation operators applied to textual notation can be used in conjunction with graphical notation so that for example a guard condition of a transition is negated using Expression negation operator.

For security and vulnerability testing, the mutation operators should be based on common programming errors likely to trigger vulnerability. For example the above mentioned mutation operators can be readily used to address vulnerabilities such as buffer overflow as the causes for such include logical errors such as off by one error, absence of a null character at the end of a buffer, use of functions that do not check the buffers sizes when performing certain operations, incorrect buffer variable values, etc. These mutation operators can be used for detecting boundary condition errors by applying for example relation operator replacements, input validation errors, access right errors, environment errors such as integer under and overflows, and synchronization errors where mutations are used to reorder the lock/unlock operations.

2.5.3 Mutations for Test Generator

Mutation analysis is usually applied to actual program code to automatically generate unit tests and like. However, mutation analysis can be also applied in specification or design model driven automatic test generation. This approach mutates the specification instead of real program code and then applies techniques to automatically design test cases to compare the mutants with the original specification to automatically design and generate tests and to evaluate coverage.

When mutation analysis is applied for design model driven automated test design, it can be used to (1) **validate** test cases and/or to (2) **generate** test cases.

Essentially when mutation analysis is used to validate test cases we first generate test cases without any mutants. Then mutation operators are applied to the model in order to generate mutants. Test cases are then executed against on each mutant and if the output of the mutant differs from the original output (the correct one), the mutant is marked as being dead.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 81 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

When mutation analysis is used to generate test cases, the test generation tool simply generates tests that kill the mutants.

After these steps, we need to identify equivalent mutants (i.e. those mutants that cannot be killed due to the fact that they are semantically equal and they produce the same output as the original system) in order to provide a precise and accurate mutation score. Finally, mutation score is calculated and presented.

2.6 MODEL-BASED MUTATION TESTING

In this section we discuss the influence of strategies for generating test cases from models using mutations. The content of the section originates from results obtained during the EU FP7 project MOGENTES (www.mogentes.eu). The reason for adding this section relies in the potential importance for DIAMONDS when using model-based mutation testing in the context of testing security issues. Let us discuss the usefulness of mutation testing in the context of security testing using a simple example from [33] (page 51). In the example, an exploit is used that allows an attacker to execute a shell command on a server. The corresponding software bug on the server side that causes this vulnerability might look like this:

```
$username = ARGV;
system("cat /logs/$username" . ".log");
```

In this program fragment a call `system()` takes a parameter (`$username`) that is unchecked. If `$username` takes an existing user, a file with name `$username` and extension `log` in the directory `/logs` is displayed. In case the user does not exist, an error is reported. Now consider the case where an attacker provides a `$username` that looks like this:

```
bracken;rm -rf / ; cat blah
```

In this case the system call would have the parameter

```
cat /logs/bracken;rm -rf /; cat blah.log
```

which removes all files on the server. In terms of mutation testing parameters that are passed to a server has to be mutated. For this purpose a mutation function has to be used that replaces the parameters with different shell scripts or commands. The underlying model would be, for example, a specification of the expected parameters or arguments, or a description of how to communicate with the server. The mutations would be possible deviations of data or communication patterns that are critical for security testing.

When considering mutation testing for security testing the question of how to efficiently and effectively creating test cases becomes an important issue. In the rest of this section exactly the question of which strategy to choose for test case generation in practice is answered in detail.

2.6.1 Introduction

In this section we introduce model-based mutation testing strategies. In this testing approach the ideas of program mutation [21] [31] are applied on the modeling level. The idea is to mutate abstract models of a system under test (SUT). Mutation operators implement the mutation process by altering the models syntactically. These mutated models represent faulty designs. Then test cases killing the mutated models are generated, i.e. we are interested in test cases that can distinguish the original from the mutated model. These tests are then executed on the SUT and will detect if a mutated model, i.e. a faulty model, has been implemented.

The advantage of this testing style for security testing is that it is fault-centered. In contrast to cover certain states or transitions in a model, we focus on possible flaws in the model. Hence, the generated tests cover flaws in the design of a system. The testing strategy is very fine grained. Whatever can be considered a fault on the modeling level can be covered by a test case.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 82 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

In the EU FP7 project MOGENTES (www.mogentes.eu) we have taken this challenge and developed a model-based mutation testing tool for UML state transition diagrams. We map UML to labeled transition systems (LTS). This mapping has been described elsewhere [44][4]. It suffices to point out that this formal foundation gives us access to the existing testing theories on LTS. We have chosen Tretmans's testing theory with the conformance relation ioco [76], because it supports partial and non-deterministic models.

We have developed a test case generator that essentially performs an ioco check. If a mutant does not conform to its original model a counter-example test case is generated. The general idea and its extension to hybrid system testing has been presented in [5]. The conformance checker itself has been discussed in detail in [18].

Most recently we have investigated different strategies for detecting for generating a test case once non-conformance has been detected. The strategies vary in the number of generated test cases, in the generation time and in the shape of the test cases. The latter refers to the fact that some strategies generate linear test cases, others adaptive test cases. We have implemented eight such test case selection strategies and compared them in two case studies. The first case study is a car alarm system. In addition to this rather small example, we analyzed a second case study on testing the control of a wheel-loader bucket arm. The results of the car alarm have been published in [6]. The second case study is unpublished.

Why so many strategies? The aim was to explore different ideas of how to generate the tests and investigate how successfully they are in finding bugs. The point is that a minimum set of shortest test cases is not necessarily the best strategy to find a bug in an SUT. Therefore, we started with the straightforward but costly strategy to cover all paths to the point of non-conformance and evolved to a strategy for generating adaptive test cases. Then, we added the ability to check if existing test cases can already kill a mutant. This resulted in a combination of random testing and conformance checking.

In order to compare our six mutation testing strategies to more traditional ways of test case generation, we added pure random testing and test case generation with hand-written test purposes to our experiments. The latter is implemented via a translation of our intermediate format to the CADP₊ tools. This gives us access to the TGV test case generator [41] as well as to model checkers and simplifiers of CADP.

In our experiments we wanted answers to the following questions:

- What is the best strategy to select a test case? •
- Is mutation testing better than random testing? •
- How efficient is the test case generation? •
- How severe is the equivalent mutants problem? •
- Do partial models help? •
- Does the combination of random testing and mutation testing help?
- Given a set of faulty implementations. Can we find all known bugs?

In the next section we present our running example.

2.6.2 A Car Alarm System

In order to demonstrate the basic concepts of our test case generation approach, we use a simplified version of a car alarm system (CAS). The example is taken from Ford's automotive demonstrator within the MOGENTES project. The following requirements were specified and served as the basis for our UML test model:

R1 - Arming The system is armed 20 seconds after the vehicle is locked and the bonnet, luggage compartment, and all doors are closed.

R2 - Alarm The alarm sounds for 30 seconds if an unauthorized person opens the door, the luggage compartment, or the bonnet. The hazard flasher lights will flash for five minutes.

R3 - Deactivation The anti-theft alarm system can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside.

As shown in Figure 34, the UML test model resulting from the above stated requirements comprises four classes and four signals. The class AlarmSystem is labeled as system under test (SUT) and may receive any of the Lock, Unlock, Close, or Open signals. At the same time, the SUT calls methods of the classes AlarmArmed, AcousticAlarm, and OpticalAlarm – all of them labeled as being part of the environment. In this way, the class diagram specifies the test interface.

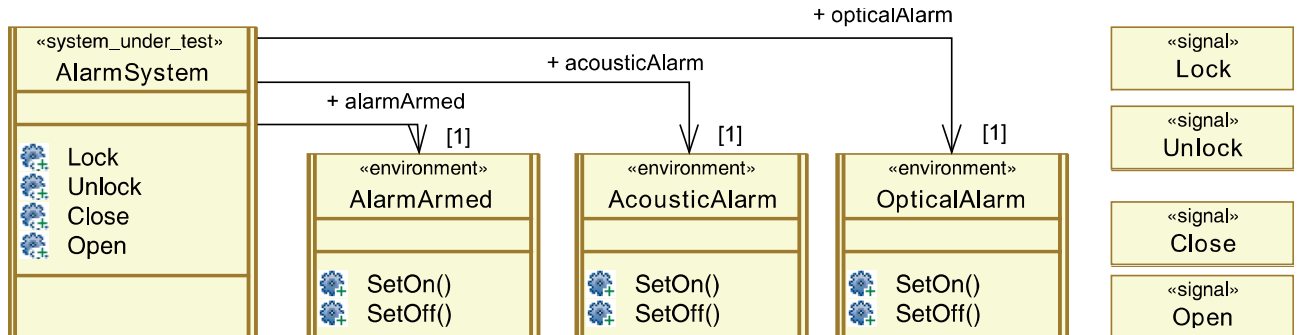


Figure 34: Test interface of the car alarm system.

Figure 35 shows our CAS state machine diagram. From the state OpenAndUnlocked one can traverse to ClosedAndLocked by closing all doors and locking the car. Actions of closing, opening, locking, and unlocking are modeled by corresponding signals Close, Open, Lock, and Unlock. As specified in requirement R1, the alarm system is armed after 20 seconds in ClosedAndLocked. Upon entry of the Armed state, the model calls the method AlarmArmed.SetOn. Upon leaving the state, which can be done by either unlocking the car or opening a door, AlarmArmed.SetOff is called. Similarly, when entering the Alarm state, the optical and acoustic alarms are enabled. When leaving the alarm state, either via a timeout or via unlocking the car, both acoustic and optical alarm are turned off. When leaving the alarm state after a timeout (cf. requirement R2) we decided to treat the underspecification in the requirements in the way that the system returns to an armed state only in case it receives a close signal. Turning off the acoustic alarm after 30 seconds, as specified in requirement R2, is reflected in the time-triggered transition leading to the Flash sub-state of the Alarm state.

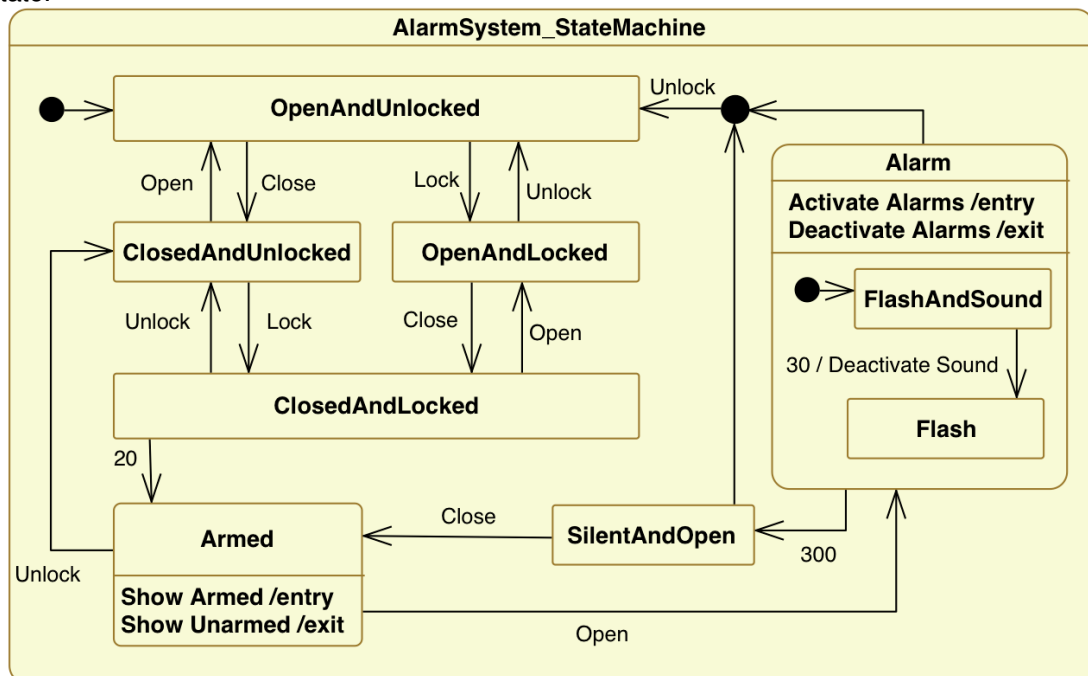


Figure 35: State machine of the car alarm system.

Since we want to create test cases that cover particular fault models, we deliberately inject 'bugs' into the specification and then create test cases that identify the differences. Therefore, we use various mutation operators. For example, one mutation operator sets guards of transitions to false. Others remove entry actions, signal triggers, or change signal events. For the CAS specification, we obtain 76 mutated UML state

machines. 19 mutants with a transition guard set to false, 6 mutants with a missing entry action, 12 mutants with missing signal triggers, 3 with missing time triggers, and 36 with changed signal events.

We create first-order mutants, i.e., each mutated state machine covers only one particular mutation (one mutation operation in a particular place). Mutation testing relies on two assumptions: (1) competent engineers write almost correct code, i.e., faults are typically “one-liners” and (2) there exists a coupling effect so that complex errors will be found by test cases that can detect simple errors.

In the following we discuss the different strategies to kill the mutants.

2.6.3 Mutation Killing Strategies

We have implemented a tool chain for checking the input-output conformance of two system models. First we translate the UML models to an intermediate model in the style of Back’s Action Systems. Then, our tool Ulysses performs a input-output conformance check between the two models. Figure 36 depicts the computation steps of our tool. Ulysses expects two labeled action systems as input: (1) a system specification AS and (2) a mutated version of the same specification AS^M . Ulysses explores the action systems and interprets the labeled actions as labeled transition system (LTS). Then this LTS is enriched by quiescence and subsequently converted into a deterministic automaton. By executing these steps, which are depicted in the first box of Figure 2.5.3, we have obtained a so-called suspension automaton.

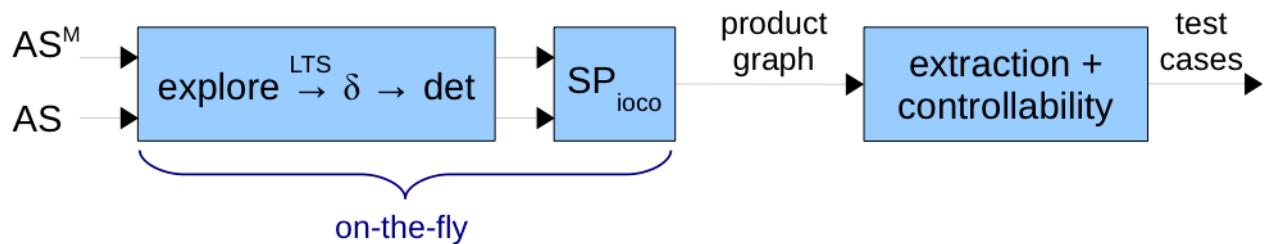


Figure 36: The computation steps of the conformance checker Ulysses.

Ulysses generates the suspension automata for both input models AS and AS^M . Afterwards, the ioco check for these two models is performed (see the central box in Figure 36), which generates a product graph from which we extract controllable test cases (last box in the Ulysses process). Note that the calculation of the suspension automata and the synchronous product calculation modulo ioco (SPioco) are performed on-the-fly, which means that the automata are only unfolded as required by the conformance check. For details see [BWA10].

In the following we compare eight different test case generation strategies to extract the test cases and name them S1 to S8. Generally speaking, S1 to S6 are fault-based approaches, S7 is based on random testing, and approach S8 uses test cases that were manually designed. All fault-based approaches use a set of 76 faulty models as input to test case generation. While the test cases in approach S1 and S2 have been generated via straightforward path search the approaches S3–S6 employ a test case generation algorithm that produces branching adaptive test cases for non-deterministic systems. Note that because the car alarm system is deterministic, all test cases are linear. In S7 we combine random and adaptive test case generation. Table 1 highlights the main differences between the strategies regarding generated test cases.

Table 1: Number of test cases for the CAS generated by our eight different approaches.

	S1	S2	S3	S4	S5	S6	S7	S8
Max. Depth	10	14	23	23	23	150 (19)	150	30
Gen. TCs [#]	16 210	302	504	129	63	11	3	9
Duplicates [#]	12 179	174	217	0	0	0	0	0
Unique [#]	3 469	110	269	123	59	11	3	9
Gen. Time [min]	188	91	23	70	23	10	0.25	-

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 85 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

Strategy S1: Within the first approach, the product graph of the ioco check is first transformed into a tree structure with a maximum depth of 10 and a two-times maximum visit per state per trace. After tree unfolding the graph, we generate linear test cases for all paths that lead to an unsafe state in the tree. As can be seen in the table, this greedy test case generation strategy produces a very large number of tests. Also, approximately 75% of the generated tests are duplicates. From the non-duplicate test cases, we could exclude another 562 tests that are a complete subsequence of one of the other remaining test cases. Hence, about 3500 tests remain. Due to the vast amount of test cases generated by S1, we refrained from executing the tests on the CAS implementations.

Strategy S2: The second test case generation strategy directly works on the product graph and extracts only one arbitrary linear test per unsafe state. Notice that one mutant may still involve more than one unsafe state in the product graph. This time, we could identify 58% of duplicate test cases and another 22 were covered as a subsequence in other test cases. It has to be said that we allowed for tests with a maximum length of 14 in this approach, thus generation times are not directly comparable between S1 and S2. We verified that all unique test cases with a length of up to 10 that were produced by this approach were included in the set of tests generated by the first approach.

Strategy S3: The third approach is similar to the second one, except that the depth is theoretically unbounded since the test case generator applies our adaptive generation algorithm. Therefore, the generation time is not comparable with approaches S1 and S2 (but with S4 to S7).

Strategy S4: The fourth strategy builds on S3 but avoids creating test cases for unsafe states in a given product graph which are already covered by existing test cases. An unsafe state is the state where the mutation (or fault) is activated. In order to check if a certain unsafe state is covered by an existing test case we generate a test purpose tp for that unsafe state. Then, the tool checks the existing test cases by computing the synchronous product with tp.

This approach yields one test case per unsafe state. Note however, that generated test cases still can be included in other test cases. This depends on the order of processed unsafe states. If we would start with deep unsafe states and proceed to shallow ones it is theoretically possible that no generated test case is included in another one.

While the total number of generated test cases decreases to 129 (from 504), the time used to generate them increases to 70 minutes (from 23). This is because we compute the check for every unsafe state and in the worst case for all test cases generated so far. In effect, S4 checks whether an existing test case already covers an unsafe state in the ioco product before creating a new test case. If an unsafe state (there may be several per mutated specification) is not covered, a new test case is emitted.

Strategy S5: Strategy S5 also avoids creating duplicate test cases. However, S5 further tries to minimize the size of the generated test suite. Before creating test cases for a mutated specification, S5 first checks whether any of the previously created test cases is able to kill the mutated specification. Therefore, the synchronous parallel execution of the test case with the mutated specification under test is applied. Hence, here we are satisfied if one unsafe state is reached, in contrast to S4 where all unsafe states will be covered. Therefore S5 rather covers mutants than unsafe states in a mutant.

Strategy S6: Approach S6 is like S5 but instead of starting with an empty test suite, S6 uses one randomly generated test case to start with. In effect, S6 is a combination of random (S7) and fault based testing (S5). Within Table 1, the maximum depth not put in brackets is the depth of the randomly generated initial test case while the figure in brackets is the maximum depth of the additionally generated tests to cover all faulty specifications.

Strategy S7: This approach uses a random selection strategy in order to generate test cases. Put differently, S7 is not a fault-based test case generation approach and included for evaluation purposes only.

Strategy S8: Finally, we compare our mutation test case generation approach with manually created test cases. More precisely, we generated 9 different test purposes by hand and let TGV [JJ05] create test cases.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 86 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

3 out of the 9 test cases check for observable timeouts (time-triggered transitions: 20, 30, 300 sec. delay). 4 test cases check the entry and exit actions of the states Armed and Alarm. One test case checks for the deactivation of the acoustic alarm after the timeout and one more complex test case has a depth of 30 transitions going once through the state SilentAndOpen to Armed before going to Alarm again and leaving after the acoustic alarm deactivation by an unlock event. Hence, each observable event is covered by at least one test case. During the creation of the test purposes, we relied on a printout of the UML state machine.

2.6.4 Test Case Execution Results for the Car Alarm System

We applied classical mutation analysis in order to evaluate the effectiveness of our different test suites. For this purpose, we have implemented the CAS in Java based on the state machine of Figure 35. In order to derive a set of faulty implementations, we used the MuJava tool [50]: in total, MuJava gave us 72 mutated implementations. After careful inspection, 8 of these mutated implementations were found to be equivalent to the original program, and another set of 26 mutants was found to be equivalent to other mutants - forming 26 equivalent pairs. Hence, a sum of 38 (72 - 8 - 26) different faulty implementations of the CAS remain. Further details are shown in Table 2.

For each method, the table lists the total number of mutated implementations, the number of mutants that turned out to be equivalent to the original implementation, and the number of equivalent pairs of mutated implementations. The methods Close, Open, Lock, and Unlock are public ones and handle the equally named external events while SetState and the constructor (Constr) are internal methods. From the table, one can observe that the mutation on internal methods has a strong effect on the external behavior since there are no equivalent mutants for these methods.

Table 2: Injected faults in the CAS implementation per method.

	Mutants	Equiv.	Pairwise Equiv.	Different Faults
SetState	6	0	1	5
Close	16	2	6	8
Open	16	2	6	8
Lock	12	2	4	6
Unlock	20	2	8	10
Constr.	2	0	1	1
Total	72	8	26	38

In the following, we use the 38 unique faulty CAS implementations to evaluate the effectiveness of our generated test cases. Of course, all tests were validated on the non-mutated CAS implementation. Table 3 gives an overview of the number of survived faulty implementations for each test case generation approach.

Table 3: Overview of how many faulty SUTs of the CAS survived the generated test cases.

	S2	S3	S4	S5	S6	S7	S8
SetState	0	0	0	0	0	0	0
Close	1	0	0	0	0	0	2
Open	0	0	0	1	0	0	4
Lock	0	0	0	0	0	0	2
Unlock	1	0	0	0	0	1	5
Constr.	0	0	0	0	0	0	0
Total	2	0	0	1	0	1	13
Detection Rate [%]	95	100	100	97	100	97	66
Impl. Coverage [%]	99	99	99	99	99	99	88

As can be seen from the table, the approaches S3, S4, and S6 were able to reveal all faults. The table also proves that the minimization algorithm applied in approach S5 reduces the fault-detection capabilities. Despite random testing (S7) did not find all faulty implementations, it proved quite effective in our setting. It has to be noted, however, that we allowed for an appropriate depth of the random test cases. Summing up, S5 and S7 still have a fault detection rate of 97%. The reason for the bad performance of S2 is the fact that the two surviving faulty implementations need tests with a depth of more than 14 interactions to be revealed and S2's tests are restricted to a depth ≤ 14 .

The last column shows the results of running the manually designed tests (S8). Overall, 25 mutants were killed which results in a detection rate of 66%. Clearly, the figures show that in order to have a meaningful test suite, more (diverse) test cases have to be generated. Partly, this lack of diverse test cases is based on the deterministic test selection behavior of TGV: all TGV based tests have almost the same test sequence from OpenAndUnlocked to ClosedAndLocked, although alternative paths are possible.

Often, coverage metrics on the implementation's source code serve as a quality measure to describe the adequacy of a test suite. Therefore, we have measured the coverage on the implementation in terms of basic block coverage. The approaches S2 - S7 achieve a coverage of 99%, whereas S8 (TGV) only results in a basic block coverage of 88%. By comparing the last two rows of Table 3, it becomes obvious that a high code coverage does not automatically guarantee high fault detection rates, which we aim for.

In summing up, the results show that our approaches are powerful in covering the implementation's source code and more importantly in detecting faults. However, the depth of our conformance analysis is critical as too less depth results in missing test cases and, in this example, in undetected faults. The results also show that the 3500 test cases of the first approach were by far too many: for the given

2.6.5 A Second Experiment: a Wheel Loader

While the Car Alarm System provided us with valuable insights during the tool development and the initial tool-evaluation phase we consider it as an example too small for generating representative results.

In a second experiment, therefore, we used the much more complex model of an ECU (Electronic Control Unit) controlling the arm and the bucket of a wheel loader.

The principle system architecture of this demonstrator is as follows. The ECU receives signals from an ISOBUS (CAN) network and controls the actuators of the bucket/arm as well as a TFT display that shows important (graphical) information about the current status of the implementation. Inputs to the ECU come from, e.g., a joystick that is installed on the wheel loader. The ECU is responsible for setting the current that flows through the electromagnets and in turn controls some valves on the actuators on the arm/bucket of the wheel loader.

Important behaviors that had to be tested were bus initialization (address claim), error management, and general timing related properties. It has to be said that this model proved to be a challenge as different low-level bus-related events had to be modeled in detail (i.e., integer- parameterized events) while at the same

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 88 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

time it also showed a high degree of parallelism (i.e., interleavings of event sequences) and complex time-out behavior.

The OOAS model (translation from UML model) for the bucket control of the wheel loader consists of 137 actions. In order to cope with the complexity of our fault-based strategies on this use case, a separation into several partial models had to be done. The splitting involved partitioning the input values coming from the joystick into equivalence classes. The first partial model is called EQC. It only uses 11 different values per axis each representing one equivalence class. Note that one of the 11 values represents an error state of the joystick. In principle this enables the error handling mechanisms of the model. Nevertheless, due to the complexity of this model we cannot explore the model up to the necessary depth to observe any effect of the error handling.

In order to reach the parts of the model, where the effects concerning the error management can be observed, a second sub-model called X error has been created. It focuses solely on error handling and is simplified even more. Instead of considering 11 different values per axis, in this model one axis has a constant value. The other axis is limited to a set of three possible values.

Table 4: Number of test cases for both wheel loader models generated by strategy S5.

Test Case Generation with S5	EQC	X_error
Max. depth for <i>ioco</i> check	13	45
Mutants [#]	220	219
Generated test cases [#]	217	20
Mutants triggering a new test case [#]	10	13
Additionally killed mutants [#]	79	144
Mutants equiv. up to max. depth [#]	131	62
Average time for generating one test case [min]	20	35
Average time for killing a mutant with an existing test case [min]	1.7	0.5
Average time for <i>ioco</i> check for mutants equiv. up to max. depth [min]	33	166

Test case generation. Table 4 presents the most important metrics describing test case generation with S5 for these two versions of the wheel loader. The EQC model could be explored up to a depth of 13. There were 220 mutated models. Ten of them were triggering a new test case, resulting in 217 test cases. These test cases were able to kill 79 of the mutated models in addition to the ten mutants they were generated from, i.e., 89 of the mutated models were killed in total. The remaining 131 mutated models had to be considered equivalent up to the exploration depth. To generate one of the 217 test cases took 20 minutes on average. The average time needed for killing one of the non-equivalent mutants with an already existing test case took about 1.7 minutes on average. For equivalent mutants, the whole *ioco* check up to the depth of 13 took 33 minutes. The respective numbers for the X_error model are listed in the rightmost column of Table 4. The data in Table 4 shows that although we used simplified models, the generation times with strategy S5 for one distinguishing test case still exceed hours if the non-conformance between the original and the mutant cannot be identified within a certain depth (13 and 45 in our experiments). That is, we suffer from the equivalent mutants problem that manifests itself in a state space explosion problem. Hence, the purely fault-based strategy (S5) is not practically for the wheel loader use case. Our first experiments with the CAS already indicated that S5 is not the most efficient test case generation strategy.

For this reason, we generated various random test suites for both versions of the wheel loader model. They vary in their size and also in the depth of the contained test cases. Table 5 lists the combinations of depth and number of test cases. We created four test suites per model version (EQC and X_error) called Rand1, Rand2, Rand3, and Rand4. Table 5 also gives information about the generation times. Compared to test case generation with S5 (see Table 4), random test case generation is fast.

Table 5: Number of random test cases for both wheel loader models (S7).

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 89 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Test Case Generation with S7	Rand1	Rand2	Rand3	Rand4
Depth	50	50	100	150
Generated test cases [#]	256	1024	128	64
Total test case generation time [hh:mm]				
EQC	3:12	12:26	4:59	4:16
X_error	3:21	13:19	7:20	6:25
Average time for generating one test case [mm:ss]				
EQC	0:45	0:44	2:20	4:00
X_error	0:47	0:47	3:26	6:01

Finally, we applied our strategy S6 to combine random and fault-based test case generation. More precisely, we chose the random test set Rand1 as a starting test set for S6 for both versions of the wheel loader model. Table 6 presents the associated data: As with S5 (see Table 4), there were 220 mutants for the EQC model. We started with the random test suite Rand1. Hence there were 256 random test cases each having a depth of 50 interactions. Again, we explored the EQC models up to a depth of 13. There were no additional test cases generated by our fault-based approach, i.e., the final test set for the EQC model resulting from S6 consists solely of the random test cases. These random tests were able to kill 160 of the 220 mutated EQC models. Hence, 60 mutated models remained to be equivalent up to the depth of 13. On average, it took about 4 minutes to kill a mutated model with a test case and about 3 hours and 20 minutes (200 minutes) to identify a mutant to be equivalent. With the purely fault-based approach S5 (see Table 4), this time was lower (33 minutes) because there were less test cases that tried to kill the mutant before starting the *ioco* check. Note that the computation times do not consider the generation of the random test suite. For the X_error model, we had a total of 219 mutated models for test case generation with S5 (see Table 4). Due to the longsome computation times, we chose 82 mutants for test case generation with S6. Again, the initial random test set was Rand1 (256 test cases of depth 50). As with S5, we explored the model up to depth 45. For the X_error model, the fault-based approach yielded 4 test cases in addition to the random tests generated from 3 mutated models. The test set consisting of random and so far fault-based generated tests were able to kill 46 mutated models. Hence, in total 49 mutants were killed and 36 were found to be equivalent up to our maximum depth. On average, it took almost 3.75 hours (226 minutes) to generate one of the 4 fault-based test cases. To kill one mutated model with any of the existing test cases took almost half an hour and to identify a mutant to be equivalent took about 5.75 hours (349 minutes) on average. Again, note that the computation times do not consider the generation of the random test suite and that equivalent in this context means equivalent up to our maximum exploration depths.

Table 6: Number of test cases for both models of the wheel leader generated by strategy S6.

Test Case Generation with S6 and initial test set <i>Rand1</i>	EQC	X_error
Mutants [#]	220	82
Random test cases [#]	256	256
Depth of random test cases	50	50
Max. depth for <i>ioco</i> check	13	45
Generated test cases (excl. random) [#]	0	4
Mutants triggering a new test case [#]	0	3
Additionally killed mutants [#]	160	46
Mutants equiv. up to max. depth [#]	60	36
Average time for generating one test case (excl. random) [min]	-	226
Average time for killing a mutant with an existing test case (incl. random) [min]	4	26
Average time for <i>ioco</i> check for mutants equiv. up to max. depth [min]	200	349

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 90 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

Test case execution. Like in the first case study we have implemented the ECU in Java. The implementation consists of approximately 450 lines of code. Again we used the MuJava tool, which gave us 1349 mutated implementations. Considering this high amount of mutants we decided to skip the analysis whether some of the mutants are equivalent either to other mutants or to the original.

Table 7 gives an overview of the percentage of killed mutated implementations for each of our twelve test suites: S5, S7 with Rand1, Rand2, Rand3, and Rand4, and S6 (each for EQC and X error). Additionally, we computed the killing abilities for our strategies when combining the test sets from both model versions (see rightmost column). Generally, the random test sets generated from the EQC model achieve a relatively good killing rate compared to the other test sets not involving random test cases. Another observation is that the combination of our partial models makes sense, since all combinations perform better than the two separate test suites from a single model. S6 did not yield additional fault-based test cases for the EQC model and hence did not improve the killing rate of S7 Rand1. However, for the X error model, S6 generated four additional test cases, which improved the killing rate by 3.6 %. The highest killing rate of 78.9 % was achieved by the combination of the two test sets generated from S6 (last row). It combines random testing, fault-based test case generation and uses both versions of the model. Note that the killing rates in this table were not corrected by removing truly equivalent mutants.

Table 7: Killing rates of generated test cases per strategy.

	EQC	X.error	Combination
S5	28.2 %	32.8 %	46.1 %
S7 Rand1	73.7 %	30.2 %	75.3 %
S7 Rand2	76.6 %	30.3 %	76.7 %
S7 Rand3	76.7 %	30.8 %	77.3 %
S7 Rand4	74.3 %	31.0 %	74.8 %
S6 with Rand1	73.7 %	33.8 %	78.9 %

2.6.6 Discussion

In the following we discuss the results of our two experiments. We structure our discussion by our original questions as stated in Section 2.5.1.

What is the best strategy to select a test case? The small CAS case study served to exclude strategies that would not scale to large models. S1 had to be given up immediately, because it resulted in too many test cases. S2 has a low detection rate with a high number of generated test cases. Furthermore, it produces only linear test cases. The adaptive test cases of S3 have a good killing rate, but the number of test cases is too high. S4 reduces this number and keeps the high killing rate, but the generation time increased. In order to decrease generation time S5 was developed. However, the killing rate suffered. Hence, S4 is a good candidate for small models like the CAS. However, for large models it is hopelessly slow. Therefore, we chose S5 as a suitable strategy and combined it with random testing resulting in S6.

Both case studies show that pure random testing (S7) is doing fine. However, by adding the test cases of an exact conformance analysis still increases the killing rate. Hence, from a killing rate perspective the combined selection strategy S6 performed best.

Is mutation testing better than random testing? The answer depends on the requirements. Random test case generation is fast and quickly reaches a high killing rate. Hence, these tests are well suited during the development of the system. We used them to program our Java implementation of the bucket controller by test-driven development.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 91 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

However, random testing results in many test cases and in some environments this is perceived as too many. For example, in embedded systems testing with timers causing long waiting times, test execution is expensive. Here redundant tests should be avoided.

In relation to mutation testing, the randomly generated tests have no relation to the fault models to be covered. Hence, it is not easy to decide how many random tests are needed. Therefore, we developed our combined approach S6, where we first generate a set of random tests that will find the easy bugs. Then for the more subtle bugs the still missing test cases are generated.

The pure mutation approaches S4 or S5 would be better in theory, but due to the complexity of their algorithms we are not able to generate deep enough tests in a reasonable amount of time. This brings us to our next question.

How efficient is the test case generation? The bucket controller case study shows that compared to random testing the mutation testing strategy is very inefficient. The reasons is the conformance checking itself and the number of mutants to be analyzed.

The conformance check has exponential run-time. Hence, the depth to be explored is limited. In case a non-conformance is detected the problem is not that severe. The average time to generate a test case that kills a mutant is 20 (EQC model) or 35 minutes (X_error model). However, if the mutant is conform up to the maximum depths, then it can take hours before the next mutant can be analyzed. Hence, we suffer from the equivalent mutants problem during test case generation.

How severe is the equivalent mutants problem? From the discussion of the previous question we see that the problem is severe. With an equivalent mutant we have to perform a full equivalence check up to a certain depth. Since the problem is NP-complete our exploration depth is limited.

Do partial models help? Yes! For testing the error management of the bucket controller, we developed a partial model (X_error) that quickly reaches the error handling functionality. With this model we could explore deeper (depth 45 vs. 13) and could generate additional test cases for the error handling functionality. The combination of these tests results in higher killing rates.

The same effect can be noticed in the pure random testing strategy R7. Combining the random tests from both models increases the overall killing rate.

Does the combination of random testing and mutation testing help? Certainly! The data shows that the combined approach S6 has a higher killing rate than the pure strategies S5 and S7. Although, the killing rate only increases slightly, the combined approach S6 provides more trust in the testing set: The qualitative advantage of S6 is that it gives a guarantee that all modeled faults are being covered. What the random tests do not cover, the conformance checker will provide.

However, the run-time analysis shows that for an equivalent mutant the additional random tests increase the test case generation time. The reason is that in R6 first all random tests try to kill the equivalent mutant, then the equivalence check has to be started. This is worse than doing only equivalence checking as in R5.

Given a set of faulty implementations. Can we find all known bugs? No. In the car alarm system we could not kill one non-equivalent implementation mutant. This was a very subtle fault in a Java switch-statement. It seems that the fault models we were testing for were not fine enough to detect this bug. The fault models are defined by the level-of-abstraction of the model and the involved mutation operators. If a fault cannot be represented by our fault models, then we have to rely on the coupling effect. In this case it did not help.

Due to the large number of implementation mutants of the bucket controller we did not analyze which implementation mutants were actually equivalent. Hence, we do not know if we killed all non-equivalent mutants and found all bugs. However, we do know that certain strategies find more bugs than others, and this was the scope of this study.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 92 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

2.7 UMLSEC

In this section we discuss how UMLsec and UMLsec-inspired elements can be added to the system to aid the purpose of deriving security test cases. We use a three-tier-web-application – much resembling the iTrust case study – as a running example to illustrate how this procedure can carry over from system design over risk analysis to generation and selection of test cases.

2.7.1 Running Example: Three-Tier Web Application

As a running example of how our methodology carries through the various phases of development, we use a standard three-tier web application. This is a somewhat generalized instance of the iTrust case study. The topology of the typical components is depicted in Figure 37.

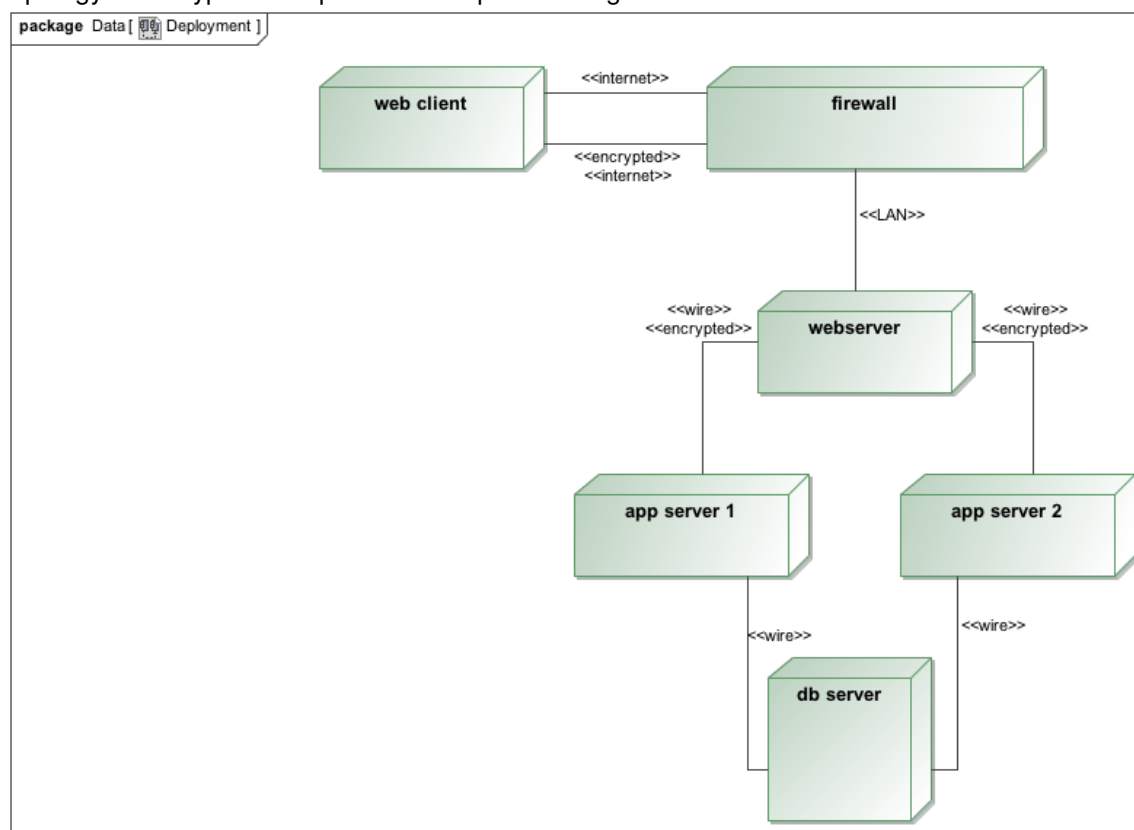


Figure 37: Topology of a three-tier Web application.

On the client side a browser is running that shows the web pages and accepts user input. In doing this browser can interpret Javascript code and hence implement some of the application logic. However, the client cannot store more information than, say, an identifying cookie. One has to also bear in mind that no security relevant logic may run at the client, since the browser is under full control of the user and may diverge in arbitrary ways from the intended behaviour. The client is connected by an unprotected and possibly by a SSL-secure internet link with the webserver. In between we typically see a company firewall that actually connects to the webserver via a LAN line. The webserver dispatches all requests to one of several application servers, which in turn are all connected to one, possibly transparently duplicated database server.

To discuss how two sample attack patterns can be addressed with our procedure, we present a subset of the risk analysis of this scenario. The attack patterns are two of the most prominent ones, SQL injection (Figure 38) and cross site scripting (Figure 39).

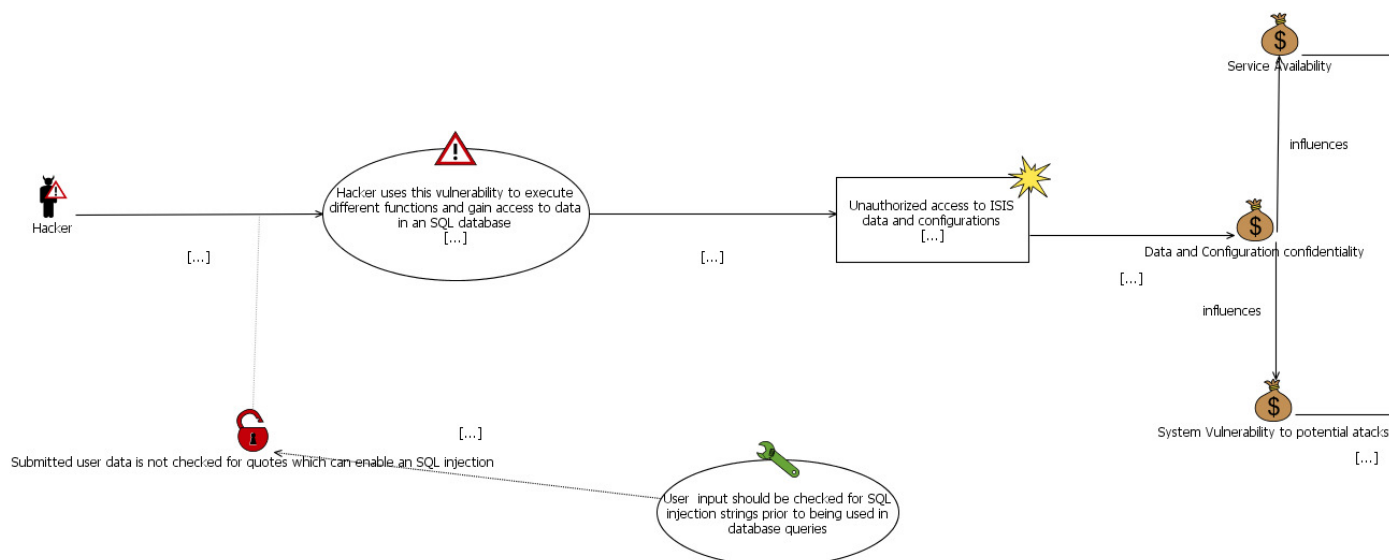


Figure 38: SQL injection risk analysis.

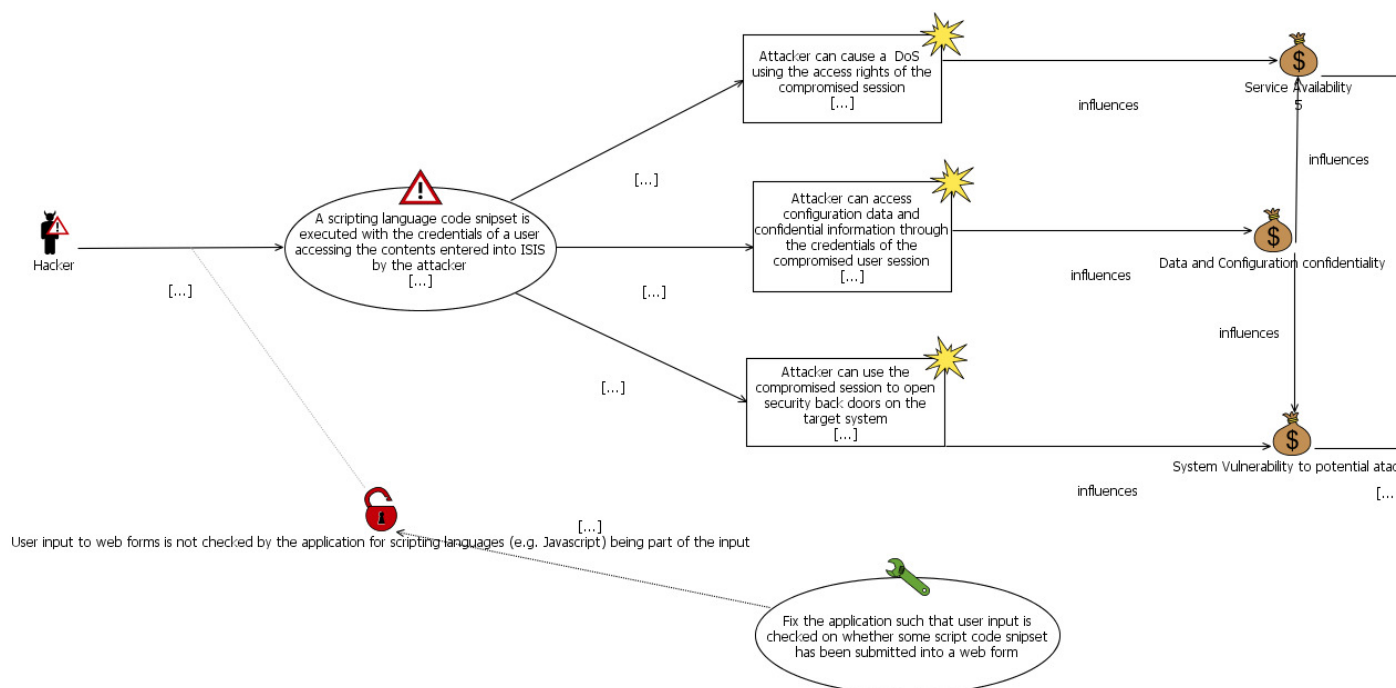


Figure 39: Cross site scripting risk analysis.

On one hand this is a suitable case study to demonstrate a sample a concrete example of our procedure. On the other we show in section 2.7.7 that we do not lose any generality and discuss how the same can be applied to other cases especially the Giesecke & Devrient case study.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 94 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

2.7.2 From CORAS Risk Analysis to UMLsec Models

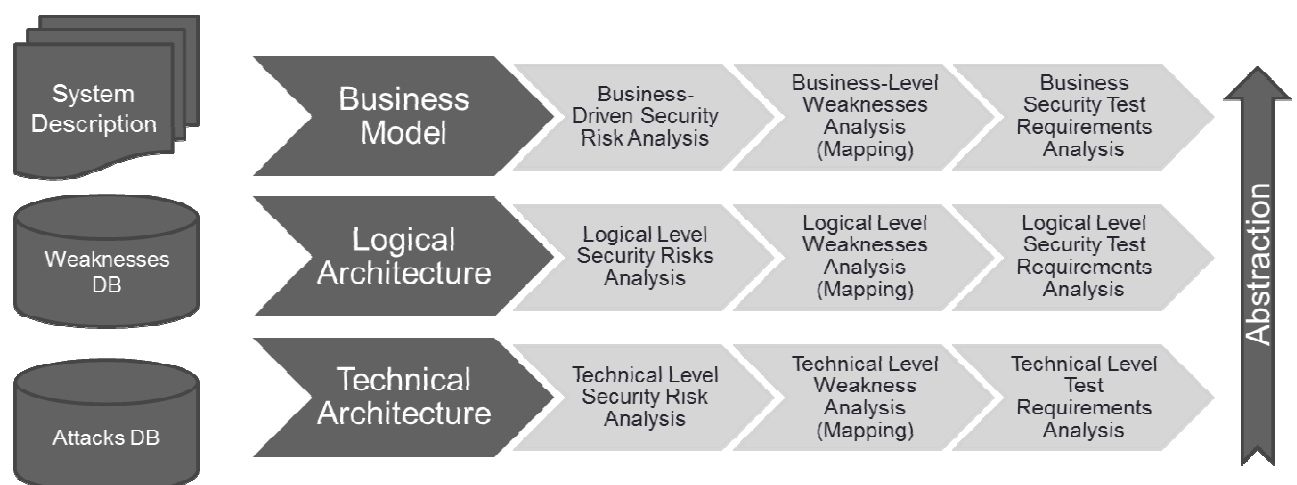
UMLSec is a UML profile defined by [43] to facilitate the annotation of UML system models with security-relevant information. The key idea is to declare security properties at design time in the system model, to allow both for model checking and test case derivation. It comprises a set of stereotypes that can be applied to various UML elements, along with tags and values to refine those stereotypes.

However, while UMLSec-enriched system models provide the appropriate information to ensure that the system built meets its security requirements, the issue of potential weaknesses and resulting vulnerabilities is hardly addressed. Therefore, to link a risk model with a UMLSec-enriched modeled the additional information must be stored either separately in a specific model or via a yet-to-be defined profile that would be applied on the system model. The purpose of that additional information is to label elements of the system architecture that are likely to introduce new weaknesses into the system or that require particular protection mechanisms. For each of those identified vulnerabilities, it will have to be ensured that proper treatment measures are elaborated and included in the system model. Additionally, it should be ensured that those treatment measures are testable, so that they will be verified on the implemented system.

It should be kept in mind that the risk analysis and the associate weaknesses analysis should be performed separately at the appropriate level of abstraction, depending of the project's development phase.

- Logical risk analysis shall be performed based on the system's logical architecture. Therefore the resulting risks may remain rather abstract and also lead to abstract risks or groups thereof.
- Technical risk analysis shall be performed based on the system's technical architecture and the resulting risks will be more specific and likely to be derived into more concrete test requirements and eventually to test cases.

Process implications



2.7.3 Generating Test Cases from the Enriched System Model

UMLsec offers UML profile element to specify security properties on several levels as depicted in (Figure 40), namely:

- Requirements/policies, e.g. Fair exchange, Secrecy, Guarded Access
- Link Types, e.g. LAN, Encrypted, Internet
- Node Types, e.g. Smart Card, POS device
- Attributes, e.g. high, critical

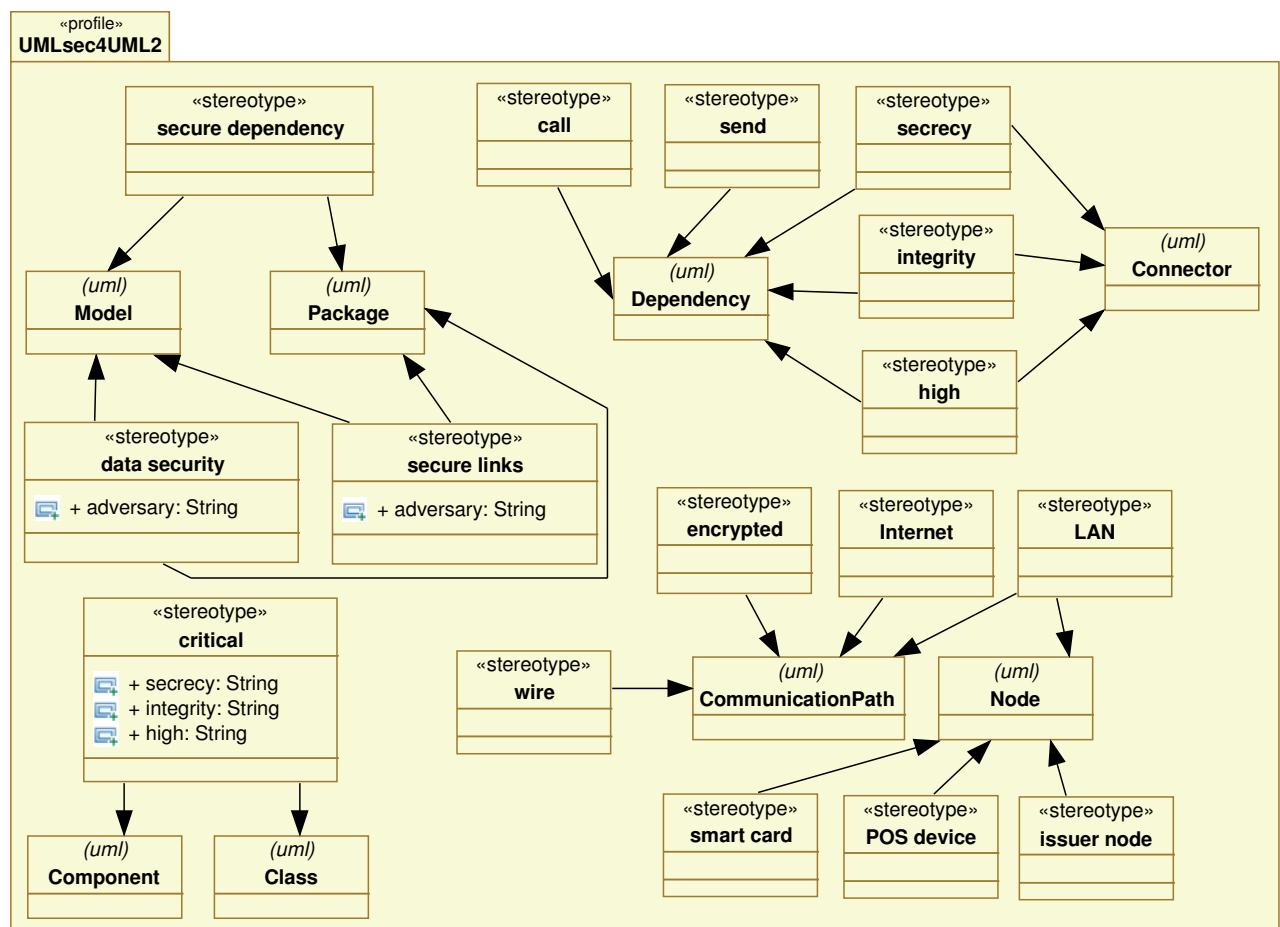


Figure 40: UMLsec stereotypes [67].

The elements are well suited and quite exhaustive to do model checking on the cryptographic security of systems. For security testing however we come to the conclusion that UMLsec must further enhanced to allow for leveraging the system model to derive test cases.

Our initial approach is to specify and track known vulnerabilities, i.e. potential weaknesses in the system design. For instance, to address attacks that follow the SQL injection pattern, we propose to add the stereotype «user provided» to mark and track any content on its way through the system. This is especially suitable for parameter of interfaces, so parameters would only be typed as String but rather as «user provided» string.

This yields useful information for the test generation phase: tests are concerned with injecting SQL codes can be guided precisely to the spots where user input is processed.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 96 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

2.7.4 Using UMLsec Stereotypes for Security Testing

In its UML profile UMLsec specifies different stereotypes that are used to define security requirements for a model. These stereotypes together with tags are used to design a system that enforces a given security policy [43]. It can then be checked if a system model fulfils its security requirements given by the stereotypes and tags. This is the domain of model checking.

For the purpose of security testing some of these stereotypes and their corresponding tags can be employed to define functional and non-functional security test cases. These security test cases can be used to determine if the security requirements are fulfilled not only by the system but by its implementation, too. There are two stereotypes and a tag that seem to be convenient for security testing: *fair exchange*, *encrypted* and *high*.

Fair exchange is used for activity diagrams and is related to the sale of goods. It requires for a certain good after passing a start node in the activity diagram a corresponding stop node is eventually reached [43], p. 55. The start and stop nodes are activities that are set in the corresponding tags *start* and *stop*. Jürjens [43], p. 56 says that the requirement given by the stereotype *fair exchange* “cannot be ensured for systems which an attacker can stop completely”. This could happen during a denial of service attack. Therefore, robustness tests e. g. fuzz testing can be used to show if this requirement is achieved by the implementation or if such an attack can be performed.

The stereotype **encrypted** says if related to a communication link in a deployment diagram the data sent along this link must be secured by cryptographic algorithms. Therefore a functional security test case can be derived that tests whether all data on this communication link are ciphered. To achieve this, more information for the test case is needed: the required cryptographic algorithm and the used keys. To extend the UMLsec profile with a new tag *cryptographic algorithm* is a possibility to make this information available from the system model. Having this information all components connected by the communication link can be tested regarding encryption using the functional security test case.

The tag *high* denotes attributes and operations to be high and can constraint the data flow when used together with the stereotype *no down-flow*. The next section describes how the tag *high* can be extended to generate test cases from state chart diagrams.

2.7.5 Exploiting UMLsec Enriched State Charts to Generate Test Cases

When creating a UMLsec model, the behaviour of a system can be specified using the different behavioural models provided by UML: activity diagrams, sequence diagrams and state machine diagrams. State machines are used here for generating test cases. To do so the *tag* specified by the UMLsec model is used and extended. The tag *high* can be used in conjunction with the stereotype *no down-flow*. It is used to declare attributes and operations as high. Operations declared as high may only rely on attributes that are also high while operations not declared as high may not access attributes declared as high. Hence it is used to separate sensitive data and access to it from non-sensitive data. No down-flow in combination with the tag *high* requires the model to respect this separation when specifying behaviour and constraints the data flow.

For test generation we can also use the tag *high*. Understanding it in a slightly different way may help for this task. Originally *high* is understood in the context of the stereotype *no down-flow* to separate data with different sensitiveness levels. To generate test cases from UMLsec state machines we need to declare not only attributes and operations as high but states, too. The idea is that certain functions may only be used after e. g. an authentication. So performing a specific operation is only allowed when reaching a specific state. Consider Figure 41: Simple state machine

showing a simple state machine for making transfer orders on a banking server. To do so an authentication is required after that a transfer order can be created. By authentication the state changes from “not authenticated” to “authenticated”.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 97 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

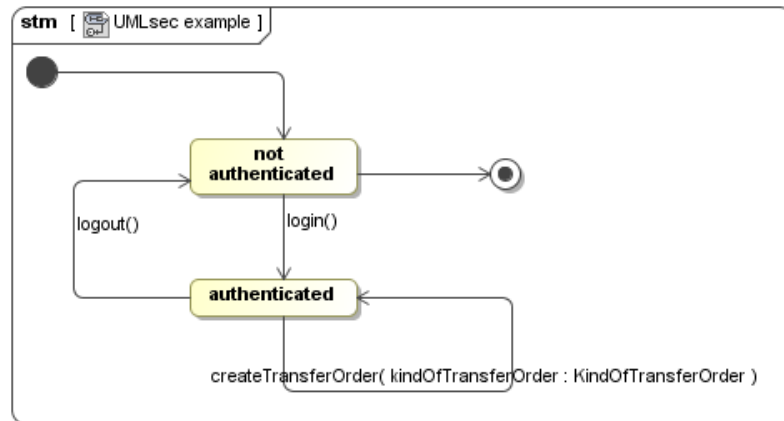


Figure 41: Simple state machine

Because the operation `createTransferOrder` may only be called in state `authenticated` it can be considered to be *high*. But this declaration itself is not useful. We need more information when it is allowed to call operations declared as *high*. To achieve this we also declare the state “authenticated” as *high* and require that *high* operations may only be called in *high* states. This results in the state machine shown in Figure 42: State machine (high operations and states are red) where high elements are red coloured. In contrast to the use of the tag *high* in combination with the stereotype *no down-flow* it does not constrain the data flow but the control flow regarding security aspects.

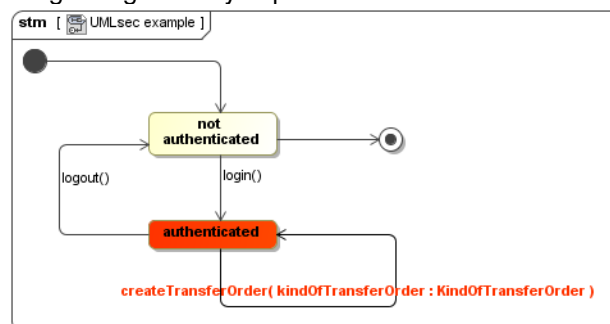


Figure 42: State machine (high operations and states are red)

The states to be *high* can be declared so by the system engineer. Another possibility is to determine the states to be high automatically. It can be said that after the execution of a certain operation all states are *high* and after the execution of another operation all states are *non-high*. If that information of operations is given, the states to be high and to be non-high can be determined automatically by traversing the state graph. After passing a transition with such an operation, the successive states are changed to *high* respectively to *non-high*. Obvious candidates for operations with those properties are `login` and `logout` operations as shown in Figure 42: State machine (high operations and states are red). They grant and revoke permissions to a system. Usually this information is only informally included in those operations but when giving it formally, the states as well as the operations to be *high* can be automatically determined. Following this approach a state is *high* if it is on a path of transitions that contains an operation that grants permissions before on that path and an operation that revokes permissions after on that path.

An operation is *high* if it is labelled only on transitions between *high* states. Operations that grant permissions when executed lead from a *non-high* to a *high* state while operations that revoke permissions lead from *high* to *non-high* states..

This state machine diagram can then be used for security testing. A corresponding test model can be generated by modifying the state machine diagram such that *non-high* operations are annotated on transitions that do not connect *high* states. In the example shown in Figure 42: State machine (high

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 98 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

operations and states are red) the operation `createTransferOrder` would be annotated on a new transition on the “not authenticated” as shown in Figure 43: State machine mutated for security testing.

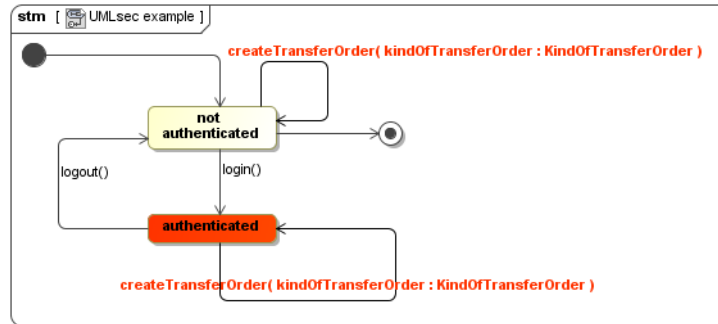


Figure 43: State machine mutated for security testing

Test cases may be generated by extracting paths from the state machine that contain transitions with *high* operations on *non-high* states.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 99 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

2.7.6 Test Case Selection and Prioritization

In order to obtain a reasonable test case selection and prioritization the information of threats given by the UMLsec model is augmented with information from a risk model designed following the CORAS method [D1.WP2 4.4.3, D1.WP3 9.2].

The CORAS method is a risk modeling approach to determine among others **threats**, for instance a hacker that may cause an **unwanted incident**, maybe overwrite entries of a database. These threats are related to unwanted incidents through **threat scenarios**, for instance “external firewall is turned off” or “hacker accesses database”. Two kinds of relations exist: **Initiates-Relations** lead from a threat to a threat scenario or to an unwanted incident. **Leads-To-Relations** lead from a threat scenario or an unwanted incident to a threat scenario or an unwanted incident. The relations, threat scenarios and unwanted incidents can be annotated with likelihoods in terms of probabilities or frequencies. The relations leading from a threat or a threat scenario to another threat scenario or unwanted incident can be amended by one or more vulnerabilities.

A simple CORAS risk model is shown in Figure 44. It consists of two deliberate human threats, a “hacker” and a “script kiddy”. These threats can use the vulnerability “SQL injection” to access a database without authorization. This is denoted by the unlocked padlock and the two relations, one leading from the threat “hacker” to the threat scenario “DB is accessible for unauthorized people” and another leading from the threat “script kiddy” to the same threat scenario. The relation from the threat scenario to the unwanted incident “Database entries are modified by unauthorized people” is another one. Except the threats and the vulnerability, all elements have probability annotations saying how large the risk that the annotated element occurs is.

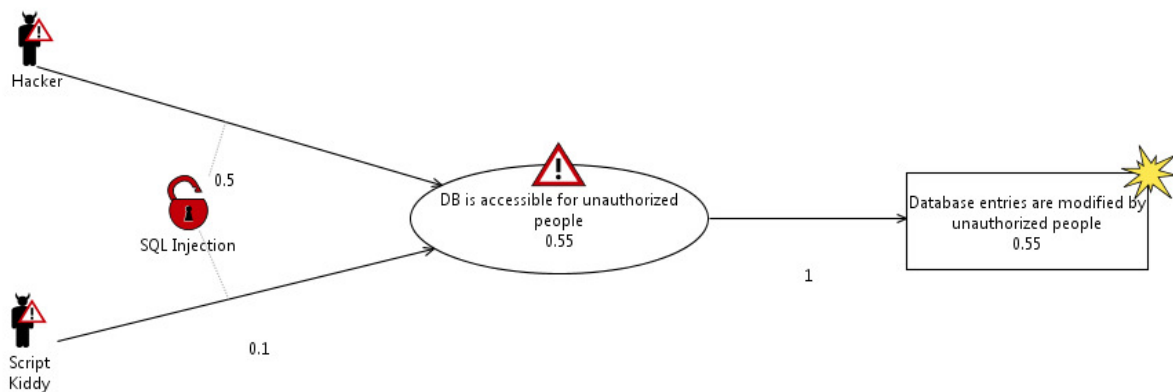


Figure 44: Simple CORAS risk model containing two threats, one vulnerability, one threat scenario and one unwanted incident

The idea is to map some of the elements of the CORAS risk model to the UMLsec system model in order to obtain risk information from the system model. These risk information can then be used for selection of components to be tested and make a prioritization of the selected components.

The CORAS risk model is used for the following purposes:

- **Determination of hazard of system components.** For this purpose the vulnerabilities mentioned in the risk model are mapped to components of the UMLsec system model. This mapping is manually done by the risk analyst. To get a probability for each vulnerability, each relation should be annotated with only one vulnerability. When a vulnerability itself is annotated on more than one relation, as above in Figure 44, the highest probability is used for that vulnerability.
- **Rating the UMLsec attacker types.** Considering only one threat in the risk model and removing all relations, threat scenarios, vulnerabilities and unwanted incidents that are not reachable from the considered threat, its dangerousness can be quantified. This is done by composing all unwanted

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 100 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

incidents and recalculating the probabilities according to the rules presented in [48], chapter 13. The CORAS threats must be mappable to the UMLsec attacker types.

The information used for test selection and prioritization is generated along the following procedure.

1. In the first step, **all vulnerabilities have to be mapped to UMLsec system components**. This is the task of the risk analyst in cooperation with the systems engineer. When mapping a vulnerability to a system component, the probability of the relation the vulnerability is mapped to the system component, too. There are two special cases:
 - If a vulnerability is related to more than one Initiates- or Leads-To-Relation with different values of likelihood (e. g. as in Figure 44 for the vulnerability “SQL injection”), the highest value of likelihood should be mapped to get the highest risk that this vulnerability is exploited.
 - If a vulnerability can be mapped to more than one component of the UMLsec model, the probability of the vulnerability should be mapped to each component without decreasing it.

The hazard of a system component in respect to its vulnerabilities can be determined by stepwise calculating the probabilities of two vulnerabilities in respect to its statistically independence or mutually exclusiveness.
2. Next, **all CORAS threats are to be mapped to the UMLsec attacker type definitions**. When mapping a CORAS threat to a UMLsec attacker type, its dangerousness should be calculated. As discussed above, this can be done by removing all model elements from the risk model that are not reachable from the specific CORAS threat. After that all unwanted incidents can be composed and the accumulated likelihood for the composition of the threat can be calculated according to the rules in [CORAS], chapter 13. As the threats are mapped to the UMLsec attacker types, its probabilities reflecting its dangerousness are mapped to the attacker types, too.
3. **Capability boundaries of the individual attacker types has to be determined using the UMLsec component diagram**. In UMLsec, each attacker has capabilities called threats regarding the different link types in a UMLsec component diagrams. For instance, an attacker type may read and write to links with the UMLsec stereotype «Internet», but only read to links with the stereotype «encrypted». If a component has two links where an attacker type has different capabilities, the attacker can obtain possibly more rights when successfully performing an attack to that component. Following the idea of trust boundaries ([70], p. 60) these capability boundaries are determined by consider each pair of link of all components and the attacker type definitions. If an attacker type actually has different capabilities on different links of the same component, this component is added to an attacker specific list of components that can be used by an attacker to get more privileges.

As a result of these steps, for each UMLsec attacker type there is a list of components that can be used to get higher privileges in a network by this attacker. These lists do not necessarily include all components of the system model. Because the hazard of each component is determined by its vulnerabilities and their probabilities, the components can be sorted by this hazard for test selection and prioritization. This is also possible for the attacker types because we calculated their dangerousness using the probabilities of the unwanted incidents they can cause.

So we can use two dimensions for test case selection and prioritization: the dangerousness of the attacker types and the hazard of each system component.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 101 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

2.7.7 Mapping to Other Case Studies: Giesecke und Devrient

The high level architecture of the banking case study resembles to a high degree the topology depicted in Figure 37. Here, too clients, webserver and database are also connected with a varying link properties and separated by a firewall. In that sense test cases may be derived from the UMLsec enriched system specification just as well as in the generalized iTrust case. However there are significant differences that must be addressed when performing the actual derivation:

- The banking scenario is much more closed in the sense that hardly any physically outside attackers have access at all to the system. Instead, risk analysis and test case derivation must focus more on maliciously incentivized employees than on the common attacker somewhere on the web or at an ISP site.
- The databases are connected on the client side rather than behind an application server. This must be considered when dealing with database related security tests.

For other case study, especially the Dornier Consulting instance, the over all architecture may be completely different, hence, applicability of this approach must be carefully evaluated on a case by case basis. Since UMLsec can, however, easily be extended further, it is very likely that the procedure can be adapted.

2.8 FUZZING UML SEQUENCE DIAGRAMS

2.8.1 Introduction

“Fuzzing is a security testing approach based on injecting invalid or random inputs into a program in order to obtain an unexpected behaviour and identify errors and potential vulnerabilities.” [17]. The aim is to find deviations of the SUT to its specification that leads to vulnerabilities because invalid input is not rejected but instead processed by the SUT. Such deviations may lead to undefined states of the SUT and can be exploited by an attacker for example to successfully perform a denial-of-service attack because the SUT is crashing or hanging.

The origin of fuzzing dated from Barton Miller, Lars Fredriksen and Bryan So [54]. They used UNIX command line tools over a modem connection in a stormy night. The stormy weather affected the phone line and so there were frequent spurious characters when using these tools. To the surprise of the authors these spurious characters caused the tools to crash and so they investigated the impact of randomly generated input values on UNIX command line utilities and doing so caused many tools to crash and/or hang.

The traditional approach of data fuzzing is generating invalid input data the SUT is fed with. This is a successful applied method to find vulnerabilities [55][27].

There are different categories of fuzzers:

1. **Random-based fuzzers** generate input data totally random. They nearly know nothing of the protocol of the SUT. Because of the usually huge size of the input space mostly invalid input data is generated [70], p. 27.
2. **Template-based fuzzers** use existing, valid traces (network traces, files ...) and modify them at some locations to generate invalid input data [76], p. 49.
3. **Block-based fuzzers** break protocol messages down into static and variable parts and generate fuzzed input data only for the variable parts. They know about field length values and checksum fields and can generate more sophisticated invalid input data [70], p. 27.
4. **Dynamic Generation/Evolution-based fuzzers** learn the protocol of the SUT from feeding the SUT with data and interpreting its responses using evolutionary algorithms or active learning [20][35].
5. **Model-based fuzzers** have full knowledge of the protocol used to communicate with the SUT. They use their protocol knowledge to fuzz data only in certain situations that can be reached by simulating the model [72].

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 102 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Following the traditional approach only input data is fuzzed. Block-based fuzzers use their knowledge about the message structure to generate new messages containing invalid data among valid data. If a message contains both valid and invalid portions of data, this amalgam is often referred to as semi-valid data [61],[70] p. 24, [27],[20]. A model can be used to generate a valid sequence of messages where just data of a few messages at certain points within the sequence are fuzzed.

Behaviour fuzzing complements this approach by not fuzzing only input data of messages but the appearance and order of messages, too. It changes the valid sequence of messages to an invalid sequence by rearranging messages, repeating and dropping them or just changing the type of message.

Behaviour fuzzing differs from mutation testing such that mutation testing in the sense of code fault injection modifies the behaviour of the SUT to simulate various situations that are difficult to test [70] p. 90,[23]. Hence mutation testing is a white box approach. In contrast (behaviour) fuzzing modifies the use of a SUT such that it is used in an invalid manner. Because the implementation of the SUT does not have to be known behaviour fuzzing is a black box approach.

The motivation for the idea of fuzzing behaviour is that vulnerabilities cannot only be revealed when invalid input data is accepted and processed but also when invalid sequences of messages are accepted and processed. For example a download may only be started after successful authentication but the download is started (by a faulty SUT) even without an authentication. In this situation the vulnerability can be exploited using valid input data but invalid behaviour when omitting the authentication.

A good real-world example is given in [70] where a vulnerability in Apache web server was found by repeating the host header in a HTTP request. This vulnerability cannot be found by fuzzing the input data. Data fuzzing would only change the parameter of the host message while behaviour fuzzing would change the number of host messages sent to the web server. Only an invalid number of host messages generated by behaviour fuzzing can reveal this denial of service vulnerability.

2.8.2 Related Work

Fuzzing is a research topic for years. There are several approaches to optimize the fuzzing process in order to generate test data that intrudes deeper in the system under test. The general problem of randomly fuzzed input data is that these data are invalid in a wide range. Because of that the input data will be rejected by the SUT before getting the chance to get deeper in the SUT [15][81][27][15]. Hence model-based fuzzing is a promising approach. Because of the knowledge of the protocol, model-based fuzzing makes it possible to get deeper in the SUT by fuzzing after passing a certain point in a complex protocol and generating invalid data only at a few points of message parameters. The model can be created by the system engineer or the tester or it can be inferred by investigating traces or using learning algorithms. There are many possibilities what can be used as a model. Context-free grammars are widely used as a model for protocol messages [15][74][2][81]. As a model for the flow of messages state machines can be employed (as in [15][2][16]) or sequence diagrams that will be later discussed.

Explicit Behaviour Fuzzing

In the PROTOS project *Security Testing of Protocol Implementations* [74], Kaksonen, Laakso und Takanen uses BNF as a context-free grammar to describe the message exchange between a client and a server consisting of a request and a response, as well as the syntactical structure of the request and the response message. The context-free grammar acts as a model of the protocol. In the first step, they replace some rules by explicit valid values. In a second step they insert exceptional elements into the rules of the grammar, e. g. artificial long or invalid field values. In the third step they define test cases by specifying sequences of rules to generate test data. Behaviour fuzzing is mentioned in [74] where the application of mutations was not constrained to the syntax of individual messages but also applied to “the order and the type of messages exchanged” [74]. Understanding behaviour fuzzing in that way random-based fuzzing implicitly performs behaviour fuzzing. Because the protocol is unknown randomly generated data can be messages and data. Hence in addition to data fuzzing also behaviour fuzzing is done – but in a random way.

For testing the IPv6 neighbour Discovery Protocol Becker et al. in [16] used a finite state machine as a behavioural model of the protocol and decomposed the messages of the Neighbour Discovery Protocol. They applied several fuzzing strategies, e. g. changing field values or duplicating fields like checksums. For applying these strategies they defined a second finite state machine as a strategic model. The combine this model with reinforcement learning defining three different reward functions. The first reward function is based on the entropy and the power of a message. The entropy of a message is defined by the distribution of

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 103 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

different function calls and the power is the amount of functions called while processing a message. The second reward function simply consists of the information whether an error at the SUT was monitored or not. The third reward function consists of the information whether corrupt or delayed messages were monitored. In [16] the different fuzzing strategies mentioned by the authors are not constrained to fuzzing input data by deleting, inserting or modifying the values of fields but supplemented by the strategies of inserting, repeating and dropping of messages that is behaviour fuzzing. Similar strategies are introduced in [35] where the type of individual messages is fuzzed as well as messages are reordered.

Banks et al. describe in [15] a tool called SNOOZE for developing stateful network protocol fuzzers. The tool reads an XML-based protocol specification containing among other things the syntax of messages and a state machine representing the flow of messages. A Fault Injector component allows modifying integer and string fields to generate invalid messages. SNOOZE can be used to develop individual fuzzers and provides several primitives to for example fuzz several values depending of their type. Among that primitives there are function to get valid messages depending on the state of a session depending on the used protocol but also primitives to get invalid messages. Thus SNOOZE enables fuzzing both data and behaviour.

The most explicit approach of fuzzing behaviour is found in [70]. Kitagawa, Hanaoka and Kono propose to change the order of messages additional to invalidating the input data to find vulnerabilities. The change of a message depends on a state of a protocol dependent state machine written in a language called *tsfrule*. But they do not describe in which way message order is changed to make it invalid.

Implicit Behaviour Fuzzing

In [81] Viide et al. introduces the idea of inferring a context free grammar from training data that is used when fuzzing for generating input data. They used compression algorithms to extract a context free grammar from the training data following the “Minimum Description Length” principle. The advantage of this approach is that the expensive task of creating a model of the SUT can be dropped and nevertheless model-based fuzzing can be conducted. The quality of the inferred model directly correlates with the amount and dissimilarity of available traces used for extracting the grammar. If the model is not exact, implicit behaviour fuzzing is done when using parts of the inferred model where differences to the behaviour of the SUT exist.

Another way of inferring a model of the SUT is by performing evolutionary algorithms. DeMott, Enbody and Punch follow this approach in [20]. They manage pools of sessions. Each session represents one full transaction with the SUT. It consists of legs that are reads or writes each containing token where a token is a piece of data. The fitness function is determined per session and per pool by code coverage of the SUT. For that purpose a modified version of the debugging framework PaiMei was brought into play. The generations are created by crossing pools, selecting, crossing and mutating sessions. After creation of a new generation the SUT is fed with the sessions of the pools and the fitness of every session and pool is recalculated. This process is stopped after a given number of generations. This is a more advanced but also not explicit way of behaviour fuzzing. Dynamic generation and evolution-based fuzzers try to learn the protocol using different algorithms as mentioned above. At the beginning of the learning process the model is mostly incorrect and so invalid messages and data are sent to the SUT. During the process the learned model is getting closer to the implemented behaviour of the SUT. During this approximation the fuzzing gets less random-based and gets subtler because the difference between the invalid generated behaviour and the correct use of the SUT gets smaller. Therefore implicit behaviour fuzzing performed by dynamic generation and evolution-based is superior to that performed random-based fuzzers.

But there is a crucial drawback of implicit behaviour fuzzing: While implicit behaviour fuzzing can find weaknesses like performance degradation and crashes, but real vulnerabilities cannot be detected. That is the case because there is no specification the revealed behaviour of the SUT can be compared to and so vulnerabilities that e. g. revealing secret data or enabling code injection are interpreted as intended features.

2.8.3 Advantages of UML Sequence Diagrams when Fuzzing Behaviour

The presented approach of fuzzing behaviour will be outlined along UML sequence diagrams. The Unified Modelling Language is a widely used standard to model object-oriented software systems and is currently available in version 2.3. It is used to define structural and behavioural aspects of software systems. One kind of a behavioural diagram is a sequence diagram. It's an interaction diagram that is used to show sequential processes between two or more objects that use messages to communicate with each other. While in object-oriented programming these messages are method calls in text oriented protocols such as HTTP these

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 104 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

messages are the verbs of the protocol. Messages may have in and outgoing parameters as well as return values. The order of messages represents their appearance in time. Figure 45 shows an example of a sequence diagram.

The goal when fuzzing sequence diagrams is to generate invalid sequences of messages. UML sequence diagrams show valid sequences of messages between two or more objects, for example a client and a server. Having defined the valid message sequences between these objects, all other message sequences are known to be invalid. Fuzzing of sequence diagrams generates these invalid message sequence using the sequence diagrams representing valid message sequences.

2.8.3.1 Combined fragments

Since UML 2 sequence diagrams can contain control structured, e. g. loops, branches, and other features like the possibility to declare not only valid sequences but also to declare a certain sequence as invalid. These control structures relate to one more certain sub sequences that are encapsulated in interaction operands of a combined fragment. Combined fragments have an interaction operator that indicates the type the fragment, e. g. *loop*, and contain one or more sub sequences called interaction operands. An interaction constraint can be defined for each interaction operand to define the conditions under that the interaction operand is executed.

For example a combined fragment with the interaction operator *loop* contains exactly one interaction operands. The interaction operand contains an interaction constraint that defines at least a value *minint* that defines the number of executions of the interaction operand. Additionally it can define an upper bound of executions defining the *maxint* value to specify a range of valid loop iterations. A Boolean expression can be specified that exhibits more constraints under which the interaction operand is executed.

Because of the additional constraints defined explicitly by interaction constraints and implicitly by the interaction operator combined fragments are helpful defining invalid sequences by violating these constraints.

2.8.3.2 State invariants

State invariants are invariants that are associated with a lifeline of a sequence diagram. They exhibit a constraint that is evaluated during runtime. If the constraint evaluates to true, the sequence is valid, otherwise it is invalid. So violating a state invariant is a way to generate an invalid sequence. But there are some limitations:

- Because of the black box nature of fuzzing the tested SUT cannot be modified. Hence only violate state invariants can be violated that are related to the lifeline of the test component.
- The constraint of a state invariant has not to refer to objects represented in the sequence diagram but can refer to external states, too. Again because of the black box nature of fuzz testing we can only state invariants can be violated that refer directly to attributes of the test component.

2.8.3.3 Time & durations constraints

Similar to state invariants time and duration constraints are evaluated during runtime and distinguish valid and invalid sequences. Sequences are valid if the time or duration constraint is evaluated to true. Like state invariants time and duration constraints can only be violated if they refer to the lifeline of the test component. But in contrast violating them is easier because the constrained element – time – is in direct control of the test component when sending messages, especially a time limit that must not exceeded.

2.8.4 General Approach

Instead of creating behaviour fuzzing test cases from UML sequence diagrams in a direct way this effort is avoided by modifying the sequence diagrams and generating new sequence diagrams by fuzzing. This approach has the advantage that already developed methods for test case generation from sequence diagrams can be used furthermore. The fuzzed sequence diagrams are generated by violating the different constraints given by it. As a result many fuzzed sequence diagrams are generated from one valid sequence

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 105 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

diagram. Depending on the complexity of a sequence diagram many different model elements can be fuzzed in many different ways.

The fuzzed sequence diagrams are generated as follow: In a first step only one model element at once leading to a fuzzed sequence diagram representing a test case. For instance, a interaction constraint of a combined fragment of the kind *alternatives* is negated. This is done for the different model elements and the possibilities to fuzz their behaviour.

In a second step, fuzzing different model elements is combined resulting in fuzzed sequence diagrams each containing at least two fuzzed model elements. For instance if a sequence diagram is fuzzed on the one hand by negating the interaction constraint of an *alternatives* combined fragment and on the other hand by repeating a single message, in the second step a fuzzed sequence diagram is created by combining these two fuzzed model element in a single fuzzed sequence diagram. This is done due to the fact that an invalid sequence containing only one invalid element does not necessarily reveal a vulnerability what is showed for data fuzzing [83].

The third step consists of fuzzing three model elements at once, for example negating the interaction constraint of an *alternative* combined fragment and repeating a message within the first interaction operand. This is done for the same reason as in step 2. The second and the third step are repeated increasing the number of fuzzed model elements in each iteration.

The number of iterations can be stopped for several reasons depending on the capabilities to get feedback from the SUT. In addition to metrics proposed in [70], e. g. interface coverage metrics or code coverage metrics, one reason to stop fuzzing could be that the SUT is rejecting all fuzzed sequences after certain iteration because they contain too much invalid messages and thus it can be expected that more fuzzed sequences will be rejected, too. This needs feedback from the SUT whether the fuzzed sequences are processed by the SUT or if they are immediately rejected by the SUT. Another possibility could be a ratio of invalid model elements and valid model elements to stop fuzzing. Eventually the process of generating fuzzed sequences should stop if nearly all model elements of a sequence diagram are fuzzed because this approximates to random behaviour fuzzing what is less efficient for data fuzzing [72].

An easy way to determine whether a test case revealed a vulnerability is valid case instrumentation [70], p. 170. Using valid case instrumentation after each fuzz test case a test case conform to the specification of the SUT is performed to determine if the SUT is still available.

2.8.5 Realization

2.8.5.1 Fuzzing behaviour of UML sequence diagram model elements

The concrete fuzzing of behaviour is realized by modifying the different model elements. This could be done in several ways:

- modifying an individual message,
- changing the order of messages,
- changing fragments of sequence diagrams,
- violating time and duration constraints as well as state invariants if possible.

The mentioned possibilities will be presented in detail in the following.

2.8.5.2 One message

Generating an invalid sequence of messages can be achieved by modifying an individual message. To obtain an invalid sequence diagram, a single message

- can be removed from the sequence diagram,
- can be repeated thus it exists twice or more,
- can be changed by type that is replacing it by another message,
- can be moved to another position,

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 106 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

- can be inserted.

2.8.5.3 Two and more messages

There are two possibilities of fuzzing two and more messages:

- If two messages are selected these messages can be interchanged. One invalid sequence can be generated this way.
- If more than two messages are selected, they can be randomly permuted. Because of its randomness this approach is less powerful than more directed ones [72]. If n messages are selected, $n!-1$ invalid sequences can be generated this way.
A less destructive approach could be rotating the selected messages. The inner structure of the sequence is only violated at three points: at the new beginning message of the sequence that was formerly not at the beginning of the message sequence, at the message that was at the end of the message sequence before rotating it and is now within the sequence succeeded by the former beginning message, and at the end of the sequence after rotating it that was formerly within the messages sequence. It tests stepwise omitting messages in the beginning of a sequence and sending it later. Following this approach if n messages are selected n different invalid sequences can be generated.

2.8.5.4 Combined fragments

Combined Fragments are control structures that describe a number of different sequences. The semantics of a combined fragment is defined by an interaction operator. It consists of that interaction operator and one or more interaction operands. Each interaction operand can have an interaction constraint as a guard. Behaviour fuzzing with combined fragments can be done in two ways: considering a combined fragment as a whole or by considering its interaction operands and interaction constraints. When considering a combined fragment as a whole, it can be fuzzed using the same mechanisms as for messages. A combined fragment

- can be removed from the sequence diagram,
- can be repeated thus it exists twice or more,
- can be changed by type that means changing its interaction operator,
- can be moved to another position,
- can be inserted.

The third and the fifth operation are – sometimes – difficult to apply to a combined fragment. In case of changing its type meaning changing its interaction operator, depending on the former and the new interaction operator more than just changing the interaction operator has to be done. When the former interaction operator is *break* that may have only one interaction operator and is changed to *alternatives* that has usually two interaction operands there is a second interaction operand to be inserted. But filling it with messages in more than a random way is difficult because there are initially no messages in the interaction operand that can be fuzzed. This is also true when inserting a new combined fragment. So removing, repeating and moving combined fragments seems to be the most useful operation when fuzzing combined fragments as whole.

The ways of behaviour fuzzing combined fragments considering its parts is discussed in the following depending on the different interaction operators defined in the UML Superstructure Specification [62].

2.8.5.4.1 Alternatives

Combined fragments with the interaction operator *alternatives* realize control structure that are known in programming languages as for instance `if ... then ... else` and `switch`. They consist of one or more interaction operands each containing an interaction constraint. The interaction constraints of all interaction operands must be disjoint. Hence at most one interaction operand is executed during one sequence.

To obtain an invalid sequence the following fuzzing operations are possible:

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 107 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

- It is possible to interchange all messages of both interaction operands. This could be done by moving all messages from the first interaction operand to the second and vice versa or by interchanging the interaction constraints of the interaction operands – either in a random manner or rotating them as described for single messages in 2.8.5.3.
- All interaction operands can be combined a single sequence and the interaction constraints as well as the enclosing combined fragment can be removed. If there are more than two interaction operands, this could be done by combining stepwise two and more interaction operands until all interaction operands are combined.

2.8.5.4.2 Option

An *option* combined fragment contains an optional message sequence whose execution is guarded by an interaction constraint. It has only one interaction operand.

An invalid sequence can be obtained by disintegrating the combined fragment and preserving the containing message sequence. However doing this, valid sequences beside invalid sequences are also generated. This is the case because the interaction constraint is removed while disintegrating the combined fragment leading to sequences of the former optional sequence that are executed if the former interaction constraint was true and was false.

Thus negating the interaction constraint is more effective because only invalid sequences can be generated from the resulting sequence diagram because the optional sequence is executed where it was not in the original sequence diagram.

2.8.5.4.3 Break

In certain situations it is necessary to perform special stuff instead of following the regular sequence diagram. This can be in case of there is an exceptional situation, for instance a resource cannot be allocated. To express this in a sequence diagram, the combined fragment *break* is used. It contains exactly one interaction operand and an interaction constraint. If the interaction constraint is evaluated to true, the interaction operand is executed and the remainder of the interaction fragment the *break* combined fragment is enclosed in is not executed.

Invalid sequences can be obtained doing the following:

- Negate the interaction operand. Doing this has the same effect as interchanging the messages of the interaction operand and the remainder of the sequence diagram.
- Disintegrating the combined fragment results – in contrast to an *option* combined fragment – only to invalid message sequences because either the interaction operand or the remainder of the enclosing interaction fragment is executed. Disintegrating the combined fragments yields to a sequence where both the sequence defined in the interaction operand and the remainder of the sequence diagram is executed.

2.8.5.4.4 Parallel

If two or more sequences of messages can be executed by merging them where only the order within one message sequence must be preserved but can be interrupted by other sequences this is expressed in a sequence diagram using the *parallel* combined fragment. The different interaction operands of this kind of combined fragments represent the message sequences whose order must be preserved but can be merged. To obtain invalid sequences, the messages enclosed in one interaction operand can be disordered. This is not only true for *parallel* combined fragments but for all messages in a sequence diagram. So *parallel* combined fragments are not helpful generating invalid sequences.

2.8.5.4.5 Weak sequencing

Weak Sequencing is the default for how the order of messages must be preserved. If nothing else is specified, weak sequencing is applied to a sequence diagram or an interaction fragment. If weak sequencing is applied, the order of messages regarding each lifeline must be preserved, the order of messages that are

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 108 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

associated to different lifelines does not need to be preserved. As a consequence, changing the order of messages generates only invalid sequences when they relate to the same pair of lifelines.

2.8.5.4.6 Strict sequencing

In contrast to *weak sequencing*, *strict sequencing* preserves the order of messages independent of the lifelines they are related to. Thus invalid sequences can be obtained by changing the order of messages without the necessity to respect the lifelines they are related to.

2.8.5.4.7 Negative

The *negative* combined fragment differs from all other combined fragments as it does not show a valid but an invalid message sequence. To obtain an invalid sequence, the combined fragment simply has to be disintegrated preserving the message sequence enclosed in the interaction operand. If there is an interaction constraint as a guard it can be negated to obtain invalid message sequences.

2.8.5.4.8 Critical region

A critical region denotes a sequence that is treated as a whole. It covers all lifelines and weak sequencing and parallel combined fragments does not influence the order of messages within the critical region. A critical region does not need to have an interaction constraint. An invalid sequence can only be obtained when the order of messages is changed or other messages are inserted in the critical region. This is not only true for a *critical region* but for all messages in a sequence diagram.

2.8.5.4.9 Ignore/Consider

Ignore and *consider* combined fragments are two sides of the same coin. Both are supplemented with a list of messages. If it is an *ignore* combined fragment, the meaning is that these messages are not relevant to determine a valid message sequence. Thus these messages can arbitrarily occur within this combined fragment without affecting the validity of the sequence. Contrariwise if it is a *consider* combined fragment only the mentioned messages are relevant for a valid sequence. Thus all other messages can be arbitrarily inserted in the message sequence.

Invalid sequences can be obtained inserting messages, change the order of messages or deleting messages that are supplementing a *consider* combined fragment. For an *ignore* combined fragment the complement of the supplementing messages can be inserted, changed in order or deleted from the sequence.

2.8.5.4.10 Loop

A loop represents a repetition of a sequence of messages. It can be set to a certain number of repetitions or limited by a lower and an upper bound. Additionally no limit can be set to tell that from zero to infinite repetitions all number of repetitions are valid.

Invalid sequences can only be obtained if there is at least one parameter for the loop:

- If there is exactly one parameter, invalid message sequences can be generated by changing it to small and greater values to test if there are off by one-issues.
- If there are two parameters, two different combined fragments can be generated one running from zero to the lower bound -1 and the other by running from the upper bound $+1$ to a maximum number.

2.8.5.4.11 Assertion

To specify that only a specific sequence is a valid continuation, the interaction operator *assert* is used. Its only interaction operand contains the valid continuation. Because of the nature of sequence diagrams representing only valid sequences – except the combined fragment *negative* – *assert* can be used in combination with the combined fragment *ignore* or *consider* to specify an exemption of the messages associated with an *ignore* or *consider* combined fragment.

To obtain invalid sequences from an *assertion* the same action can be applied to it as for *strict sequencing*.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 109 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

2.8.5.5 Time & duration constraints

Time and duration constraints can be used to specify a relative point in time where a message has to be sent or should be received or the amount of time that may elapse between two messages. Time and duration constraints can be given in different situation but only in a few of them they can be violated. There are two conditions that must be met to make a violation of a time or a duration constraint possible:

- The constraint has to be on the side of the test component. If this is not the case, the constraint has to be maintained by the SUT that is not under control of the test component.
- The occurrence(s), the constraint is related to, have to be in hand of the test component. This is for the same reason as the first condition.

If these conditions are met, the value of the constraint can be simply changed to generate invalid sequences. The fuzzed constraints must then be respected at test generation time to ensure the original constraint is violated.

2.8.5.6 State invariants

State invariants can specify many different constraints on the participants of an interaction, e. g. values of attributes or internal or external states. As for time and duration constraints state invariants has to be on side of the test component and under control of it. Because of the black box character of the presented approach only a few of the many different kinds of constraints that can be expressed by a state invariant can be used for fuzzing. These include the valuation of attributes, but not references to external states.

Another challenge when fuzzing state invariants is that just modifying them does not ensure an invalid sequence. If for example the state invariant refers to an attribute of the test component that should have a specific value, by just changing the specified value does not lead to an invalid sequence in terms of the original state invariant. In fact the behaviours that happen before the state invariant has to be changed in order to achieve the fuzzed state invariant. Additionally the value of the attribute of the test component is not under immediate control of the test component because the attribute gets its value by the SUT. Thus it is difficult to use state invariants for fuzzing.

2.8.5.7 Basic fuzzing operations

When looking at the different model elements of sequence diagrams and their corresponding fuzzing operations, commonalities between these fuzzing operations are conspicuous. For example when considering the combined fragments *alternatives* and *option* the fuzzing operations performed to obtain invalid sequences are in both cases to negate the interaction constraint. The following table composes all fuzzing operations and maps them to the model elements of UML sequence diagrams.

Fuzzing Operation	Applicable to	Generates no invalid sequences when...
Remove	single message, in all combined fragments except Negative	...applied to messages enclosed within a consider/ignore combined fragment where it is not mentioned resp. mentioned in the list of messages.
Repeat		
Move		
change type		
Insert	all combined fragments	
permute messages regarding single lifeline	≥2 messages, in all combined fragments except Negative	...applied to messages enclosed within a consider/ignore combined fragment where it is not mentioned resp. mentioned in the list of messages.
rotate messages regarding single lifeline		
interchange messages		
rotate messages	> 2 messages	

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 110 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

Fuzzing Operation	Applicable to	Generates no invalid sequences when...
permute messages regarding several lifelines	≥2 messages	
rotate messages regarding several lifelines	combined fragments of all combined fragments with only one interaction operand	
negate interaction constraint	≤2 interaction operands of combined fragments ▪ Option ▪ Break ▪ Negative	
interchange sequences of interaction operands	combined fragments with ≥ 2 interaction operands	
interchange interaction constraints of interaction operands		
disintegrate combined fragment, remove interaction constraint and possibly merge interaction operands	combined fragments ▪ Alternatives ▪ Break ▪ Negative	
change bounds of loop	combined fragment ▪ Loop	...bounds are narrower than the original bounds if two bounds are given.
change time or duration constraint	time constraint duration constraint	

Table 8: Basic Fuzzing Operations

2.8.6 Example: Application of fuzzing operations to a UML sequence diagrams

In the following a simplified example of the banking domain is used to illustrate how fuzzing operations are applied to a sequence diagram. For ease of understanding most parameters are omitted.

Figure 45 shows a sequence diagram describing how a bank customer can perform a transfer order. The customers can make national and international transfer orders. When a bank customer creates a transfer order (message 1), this is specified by the parameter `kindOfTransferOrder` have either the value `NationalTransferOrder` or `InternationalTransferorder`. In the next steps the customer send the name of the recipient of the transfer order and the amount to be transferred (message 2) as well as the recipient's national bank account information (message 3) in case of a national transfer order or recipient's international bank account information (message 4) in case of an international transfer order. The distinction between a national and an international transfer order is expressed using the combined fragment *alternatives* evaluating the parameter `kindOfTransferOrder` given by message 1. After that the transfer order must be authorized by the customer sending a valid transaction number TAN (message 5). If the customer accidentally sent an invalid TAN e. g. by mistyping it, he can try to enter a valid TAN up to two times again (message 7, combined fragment *loop*).

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 111 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

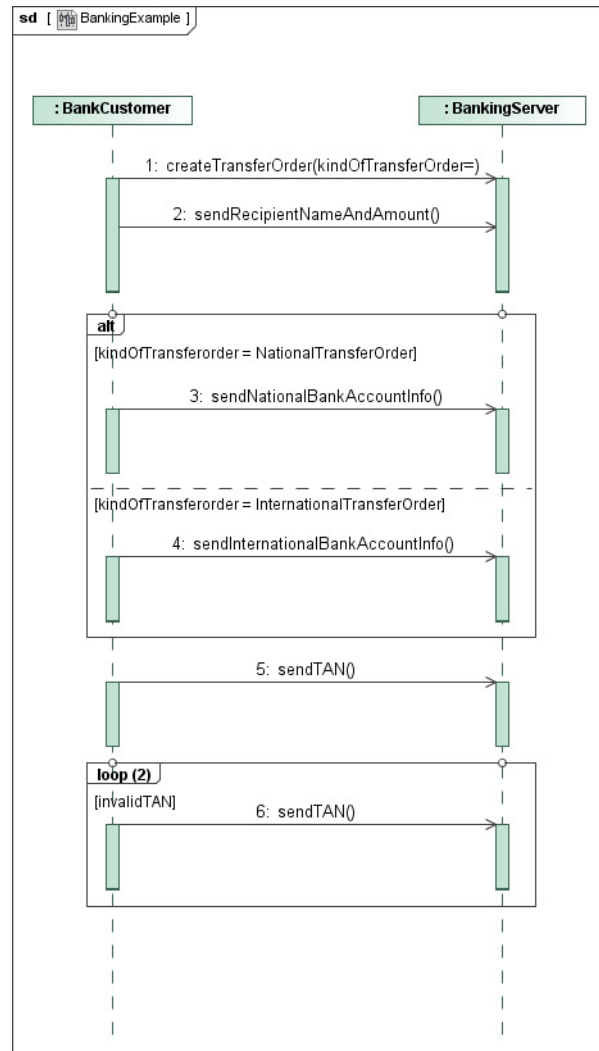


Figure 45: Simplified banking example

According to Table 8 messages everywhere in the diagram – except regarding *negative* combined fragments – can be moved, removed, repeated, inserted or the type of a message can be changed to obtain an invalid sequence. For instance message 5 can be moved to after message 2. Figure 46 shows the beginning of the fuzzed sequence diagram. The rest of the diagram remains untouched.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 112 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

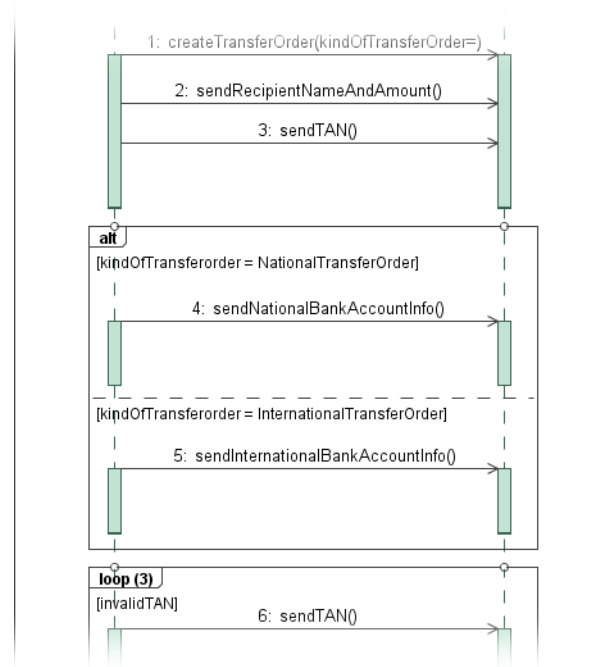


Figure 46: Invalid sequence diagram obtained by moving a message

Another possibility to obtain an invalid sequence is to negate interaction constraints of interaction operands. There are three interaction constraints: two in the *alternatives* combined fragment and another one in the *loop* combined fragment. By negating the interaction constraint of the *loop* combined fragment, the sequences generated from the resulting sequence diagram contain at least two valid transaction numbers sent to the banking server. Figure 47 shows the beginning of the fuzzed sequence diagram.

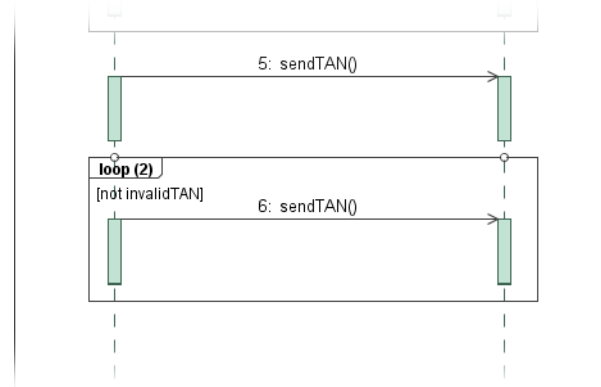


Figure 47: Invalid sequence diagram obtained by negating an interaction constraint

Also the boundaries of *Loop* combined fragments can be altered. The number of loop iterations can be changed to 3. The resulting sequence diagram – shown in Figure 48 – represents the situation where a customer can try to enter a valid transaction number up to three times after entering an invalid transaction number.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 113 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

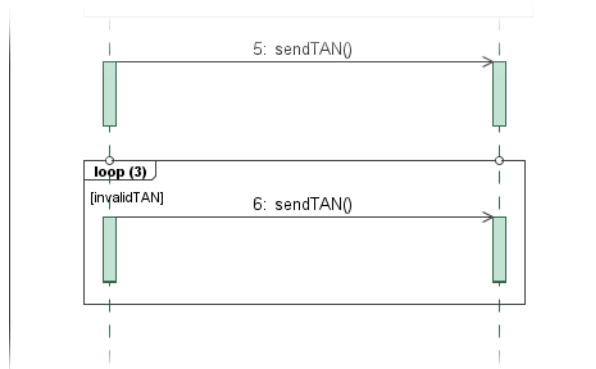


Figure 48: Invalid sequence diagram obtained by changing the bounds of a loop combined fragment

2.8.7 Concatenation of basic fuzzing operations

Performing the above mentioned fuzzing operations lead to three different sequence diagrams test cases can be generated from using known test case generation methods for sequence diagrams. As discussed in chapter 2.8.4, test cases are not only generated by applying a single fuzzing operation to any model element of a sequence diagram but also by applying a chain of several fuzzing operations.

Considering the three fuzzing operations applied to the sequence diagram in 2.8.6, we can generate fuzzed sequence diagrams by applying two of them to the sequence diagram resulting in three fuzzed sequence diagrams and by applying all three fuzzing operations resulting in another sequence diagram. Applying all three fuzzing operations leads to the fuzzed sequence diagram shown in Figure 49.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 114 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

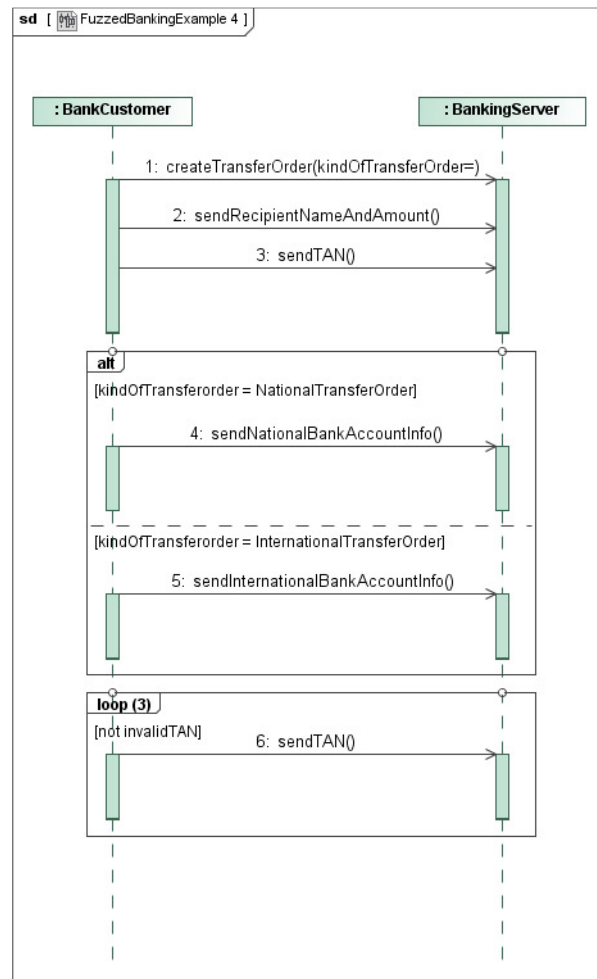


Figure 49: Fuzzed Sequence Diagram by Applying Three Different Fuzzing Operations

But applying all possible combinations of fuzzing operations to a sequence diagram cannot only lead to a large number of test cases, maybe too large to execute all of them, but also to valid sequences while invalid sequences should be generated. An obvious example is the fuzzing operation “negate interaction constraint”. When applying this fuzzing operation two times to the same interaction constraint, the result is the original interaction constraint. So this way no invalid but valid sequences are generated.

This is also true for less obvious situations. When disintegrating a *negative* combined fragment and later changing the order of messages that were formerly within this combined fragment the invalid sequences generated from disintegrating the combined fragment is then valid again. This is because the sequence specified within a *negative* combined fragment is the only invalid sequence so changing them generates valid sequences.

Sometimes it is not possible to generate sequences by applying a fuzzing operation such that the resulting test cases represent always invalid sequences. Considering the *loop* combined fragment of the example showed in Figure 45. When changing the loop parameter from 2 to 1, some resulting test cases represents invalid sequences while others do not. This is caused by the interaction constrained that additionally influences the execution of the loop. When there is first given an invalid transaction number and then a valid one, the fuzzed loop results in a test case that represents a valid sequence. Only when an invalid transaction number is sent to the banking server two times, an invalid sequence is generated. So it does not only depend on the fuzzed behaviour but also on the test data if invalid sequences can be generated from a fuzzed sequence diagram.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 115 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Another situation where invalid sequences are not generated in all cases from fuzzed sequence diagrams are branches. Branches can be realized in UML sequence diagrams by the combined fragments *alternatives*, *option* and *break*. To ensure that a fuzzed branch is used to generate test cases representing invalid sequences the interaction constraint of the fuzzed interaction operand should be set to true:

- In case of an *option* combined fragment the fuzzed sequence inside the combined fragment can be prevented from used for test case generation when its interaction constraint is evaluated to false. Hence setting the interaction constraint to true guarantees it is always used for test case generation.
- In case of an *alternatives* combined fragment the interaction constraint enclosed in the not fuzzed interaction operand should be set to false while the interaction constraint within the fuzzed interaction operand should be set to true. This maintains that all interaction constraints of the combined fragment are disjoint and ensures the fuzzed interaction operand is used for test case generation.
- In case of a *break* combined fragment, the fuzzed branch can be within the combined fragment or behind that.
If the fuzzed branch is behind the *break* combined fragment, the execution of the fuzzed sequence is prevented when the interaction constraint of the *break* combined fragment is evaluated to true. Hence it should be changed to false to ensure the fuzzed sequence is used to generate test cases.
If the fuzzed branch is inside the *break* combined fragment, it should be set to true to ensure it is always used for test case generation.

The described procedure is illustrated along the banking example showed in Figure 45. Consider the case that only the second interaction operand of the *alternatives* combined fragment is fuzzed, for example message 4 `sendInternationalBankAccountInfo` is repeated once as showed in Figure 50. If a test case is generated from this fuzzed sequence diagram that creates a national transfer order, only the unfuzzed interaction operand of the *alternatives* combined fragment is executed and thus only valid sequences are generated. To prevent this, the interaction constraint of the first interaction operand that is not fuzzed is set from `kindOfTransferOrder=NationalTransferorder` to false. The interaction constrained of the second interaction constraint that is fuzzed is set from `kindOfTransferOrder=InternationalTransferOrder` to true. The result is showed in Figure 51. This procedure ensures that in all cases only invalid sequences are generated.

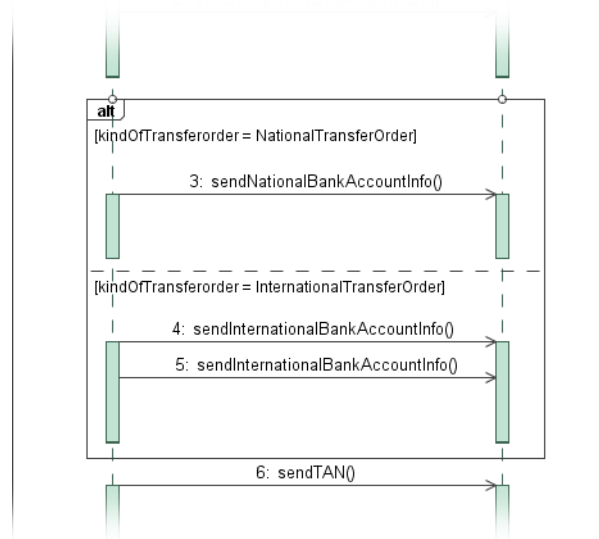


Figure 50: Alternatives Combined Fragment with Fuzzed Second Interaction Operand

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 116 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

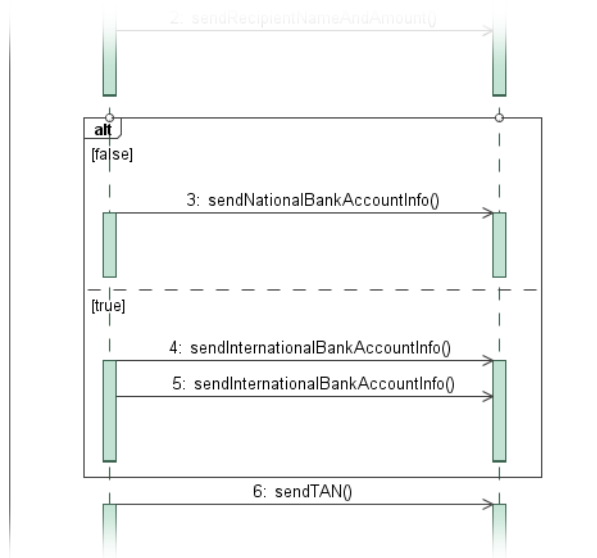


Figure 51: Modified *Alternatives* Combined Fragment Ensuring Only Invalid Sequences Can Be Generated

2.9 CONCLUSION

The introduced methods for active testing basically apply ideas from functional testing to the security domain. They are promising. Further integration seems to be necessary. The integration of test case generation strategies to the use of mutations for extracting security relevant test cases is one example. Moreover, it would also be of interest to use the obtained case studies for explaining the methods in more detail.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 117 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

3. RISK ANALYSIS FOR RISK BASED TESTING

Risk assessment methodologies like ETSI TRVA [24], CVSS [50], STRIDE/DREAD [31], OCTAVE [4], FAIR [41] and Trike [66] may help to capture risks and the risk driving factors systematically but are often unspecific on how to measure the individual factors. The main purpose of these kinds of risk analysis methods is to provide systematic process and the definition of a consistent and unambiguous vocabulary for risk identification and handling. In this regards the CERT provides taxonomy on operational cyber security risks [40]. The taxonomy identifies sources of operational cyber security risks and organizes them into four classes. It distinguishes between risks established by actions of people, by systems and technology failures, by failed internal processes, or by external events.

In Section 3.1 we mirror risk analysis techniques with respect to the requirements of the telecommunication domain, in Section 3.2 we provide an overview on the relationship between risk analysis concepts and testing concepts or model based testing concepts for risk based testing and in Section 3.3 we provide test selection techniques on basis of risk analysis results.

3.1 MIRRORING RISK ANALYSIS TECHNIQUES TO TELECOM USE CASE

3.1.1 Brief Overview of the Current Risk Assessment Process

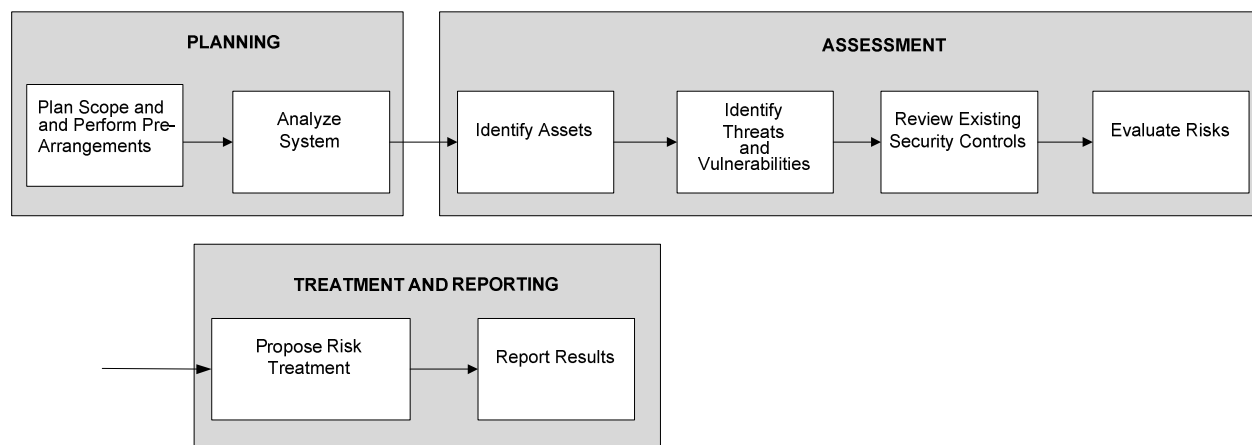
For any telecom use case (e.g. a node, product, solution), a product related risk assessment is run. This risk assessment is aligned with well know standards e.g. ISO/IEC 17799, ISO/IEC 27001, COBIT and ISO 13335.

A high level risk evaluation is done at an early stage to enable early impact on the specification of the product. For a new release of an existing product, risk assessments made for the preceding release can be built upon, thereby focusing on the impact of the incremental changes in the new release.

The following process phases are usually included in the risk assessment process:

- Planning and Scoping, including quick system analysis
- Actual risk assessment workshop addressing the following items
 - Identify Assets
 - Identify Threats and Vulnerabilities
 - Review Existing Security Controls
 - Evaluate Risks
- Propose Risk Treatments and Report Results

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 118 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public



3.1.2 Mirroring Techniques to Telecom Use Case

In order to enhance both the current risk assessment process and the actual security testing following later in the development cycle, the purpose is to study and "mirror" risk techniques as is from other consortium partners to the telecom use case. In practice the objective is to try to embed these other techniques to existing risk assessment process. The expected output is to have a separate section in risk treatment plan that is direct input to security testing planning.

3.2 TRACING BETWEEN RISK MODELLING ELEMENTS AND TEST ARTIFACTS

Risk assessment methodologies like ETSI TRVA [24], CVSS [50], STRIDE/DREAD [31], OCTAVE [4], FAIR [41] and Trike [66] may help to capture risks and the risk driving factors systematically but are often unspecific on how to measure the individual factors. The main purpose of these kinds of risk analysis methods is to provide systematic process and the definition of a consistent and unambiguous vocabulary for risk identification and handling. In this regards the CERT provides taxonomy on operational cyber security risks [40]. The taxonomy identifies sources of operational cyber security risks and organizes them into four classes. It distinguishes between risks established by actions of people, by systems and technology failures, by failed internal processes, or by external events.

In Section 3.1 we mirror risk analysis techniques with respect to the requirements of the telecommunication domain, in Section 3.2 we provide an overview on the relationship between risk analysis concepts and testing concepts or model based testing concepts for risk based testing and in Section 3.3 we provide test selection techniques on basis of risk analysis results.

3.2.1 Overview on Risk Analysis Concept

From the process point of view, the risk assessment methodologies like Risk assessment methodologies like ETSI TRVA [24], CVSS [50], STRIDE/DREAD [31], OCTAVE [4], FAIR [41] and Trike [66] have differences in detail but mainly propose the same basic actions namely:

- identification of assets,
- threat analysis,
- vulnerability analysis,
- the identification of mitigation strategies, and
- the quantification or qualification of risks

In this sense the ISO 27000 [31] definition "security risk: the potential that a threat will exploit a vulnerability of an asset or group of assets and thereby cause harm to the organization" is perfectly aligned with the activities above.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 119 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

Risk modelling tools and methodologies like fault tree analysis (FTA) [31], cause-consequence analysis (CCA) [58] or the CORAS risk modelling language [48] are dedicated to identify the risk factors and their interrelationships. During a FTA the analyst starts with a high-level fault and decomposes the fault top-down to basic events, which can be identified in the system. A variant of fault trees are so called attack trees [52]. Attack trees are directly related to security risk analysis. An attack tree starts from a complex high-level attack scenario and decomposes the attack scenario to more concrete basic interaction with the system. ETA (event tree analysis) works bottom-up, the analyst starts with the identification of unwanted system events and analyses the consequences in case of an occurring unwanted event. As a connection of both methods a CCA (Cause-consequence analysis) can be used. The analysis starts from a thread. The causes (top-down) and the consequences (bottom-up) will be analysed simultaneously.

The CORAS language integrates different tree based approaches to risk modelling. It is a graph based modelling approach that emphasize on the systematic modelling of threats or so called threat scenarios and provides formalisms to annotate the threat scenarios with probability values and formalisms to reason with these annotations. CORAS is a language in that sense, that it defines a number of risk analysis concepts with their structural interrelationship (abstract syntax), the meaning of the concepts and their interrelationship by means of a mapping to English prose (semantics), and different representation formats (concrete syntax). The main threat analysis concepts in CORAS are:

Table 9: CORAS main threat analysis concepts (see [48])

Threat	A threat is a potential cause of an unwanted incident. ¹⁶
ThreatScenario	A threat scenario is a chain or series of event that is initiated by a threat and that may lead to an unwanted incident. ¹
UnwantedIncident	An unwanted incident is an event that harms or reduces the value of an asset.
Asset	An asset is something to which a party assigns value and hence for which the party requires protection. CORAS distinguishes direct assets and indirect assets.
Vulnerability	A vulnerability is a weakness, flaw or deficiency that opens for or may be exploited by, a threat to cause harm to or reduce the value of an asset.
TreatmentScenario	The implementation, operationalization or execution of appropriate measures to reduce risk level.

Moreover CORAS allows the annotation of elements with risk estimation values and with values characterizing the consequences of unwanted incident. These concepts are summarized in the table below.

Table 10: CORAS main risk estimation concepts (see [48])

Likelihood	The frequency or probability of something (Threat scenarios, Unwanted incidents) to occur
Consequence	The impact of an unwanted incident and its consequence for a specific asset
Risk	The likelihood of an unwanted incident and its consequence for a specific asset
RiskLevel	The level or value of a risk derived from its likelihood

¹⁶ The term thread is ambiguous here. Other methods distinguish threat agents and threats where the threat agent corresponds to the CORAS::Threat and the threat to a CORAS::Threat Scenario.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 120 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

In [48] a comprehensive meta-model is given that defines the basic structural dependencies between the different risk modelling concepts.

3.2.2 A Minimal Set of Requirements Analysis and Architectural Design Concepts

Architectural software design, also described as strategic design, is an activity concerned with global requirements that deal with the general set up of a computer system. Regarding security risk analysis we are principally interested in locating critical functionality with respect to the overall software architecture and in identifying critical interfaces, which might be an entry point for an adversary. Moreover we need a general concept that describes the expectations on a system.

In the following we introduce the general concept of a requirement that describes the conditions or capabilities of system or component. We additionally introduce basic architectural design concepts like components and interfaces that are used to describe the system setup and its entry points.

Table 11: A minimal set of requirements analysis and architectural design concepts

Requirement	A condition or capability needed by a user to solve a problem or achieve an objective that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document (after [IEEE 610]).
Component	A software component is a software package, or a software module that encapsulates a set of related functions (or data). It follows the principle of separation of concern and information hiding and communicates with other components via <i>interfaces</i> . A component itself may consist of other components.
Interface	An interface is a point of interaction between components. Interfaces realize complement the principle of information hiding o components. Components using interfaces to communicate with other components via an associated protocol.

3.2.3 Overview on (Model Based) Testing Concepts

Software testing is an experimental approach of validating and verifying that a software system meets its requirements and works as expected (functional testing) or shows certain characteristics (non-functional requirements). This is done by defining test cases that are systematically directed in finding faults in the implementation and by providing test suites, i.e. specifically compiled collections of test cases, which provide an argument for the absence of faults.

Security testing is a special kind of testing with the aim in validating and verifying that a software system meets its security requirements and the security functionality works as expected (functional security testing) or that the software shows certain security characteristics. We can distinguish two different approaches. Security functional testing aims for certain functional security measures and validate and verifies the expected security functional behaviour and security tests that aim for finding vulnerabilities in a system by simulating attacks and other kinds of penetration attempts, so called penetration tests. Most known vulnerabilities originate from software faults or design flaws. However, not every fault or design flaw constitutes vulnerability.

Testing as well as security testing follow a series of activities and artefacts that aim in systematically plan, specify, realize execute test and evaluate the test results. Model based testing shows slightly different definition of these activities.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 121 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

1. *Identification of test objectives and methods:* The test objectives define the overall goals of the model based testing process and relate these goals to testing methods that allow for accomplish the objective. For model based testing the modelling, test selection and test generation strategies need to be planned.
2. *Design a functional test model:* The test model represents the expected operational behaviour of the SUT or the system environment or usage. Standard modelling languages such as UML can be used to formalize the points of control and observation of the SUT, the expected dynamic behaviour of the system, the entities associated with the test, and test data for various test configurations. The test models must be precise and complete enough to allow automated derivation of tests from these models.
3. *Determine a test generation criteria:* Usually, there are an infinite number of possible tests that can be generated from a model, so that test designers choose test selection criteria to limit the number of generated tests to a finite number by e.g. selecting highest-priority tests, or to ensure specific coverage of system behaviours. A common approach for test selection is based on structural model coverage, i.e. determining the coverage of model elements by generated tests. Another useful kind of test generation criteria ensures that the generated test cases cover all the requirements, possibly with more tests generated for requirements that have a higher risk levels. In this way, model-based testing can be used to implement a requirements-oriented or a risk-driven testing approach.
4. *Generate the tests.* The test generation is in MBT typically a fully automated process to derive the test cases from the test model as determined by the test generation criteria. The generated test cases are sequences of high-level events or actions to or by the SUT, with input parameters and expected output parameters and return values for each event or action. If needed, the generated tests are further refined to a more concrete level or adapted to the SUT to support their automated execution.
5. *Assess the test results.* During test assessment or test evaluation the quality of the SUT is rated with respect to the test results and the quality of test themselves (i.e. the fault revealing capability of the test). After the test assessment the test result may impact the system and the system requirements and the test specification.

Figure 52 shows the relationship between model based testing artefact as actually seen by the ETSI.

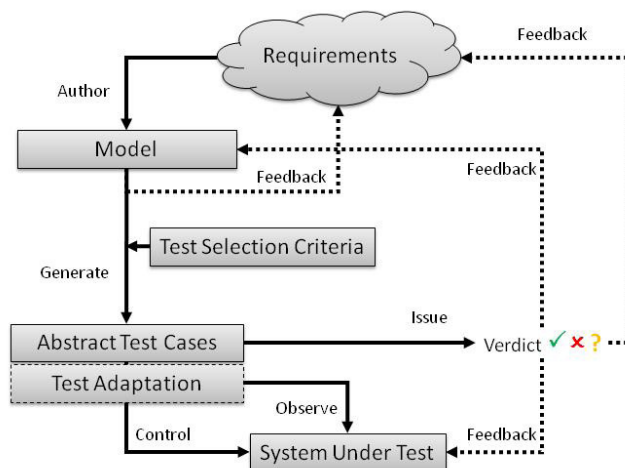


Figure 52: The model-based testing elements [81]

However, the risk based testing approach in diamonds needs to integrate in model based testing approaches that aim for test case generation and for more classical approaches that aim for the manual specification of individual test cases (either by models or by other kind of test specification or test scripting languages). The following table lists the main testing concepts that are in our opinion necessary, to describe the relationship to risk based testing.

Table 12: Overview on (model based) testing concepts

TestObjective	A reason or purpose for designing and executing a test (after [39]).
TestCase	A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement (after [37]).
TestSuite	A set of several test cases for a component or system under test, where the post condition of one test case is often used as the precondition for the next one (after [39]).
TestSelection Criterion	A property that is satisfied by a set of test cases generated from a model (after [39]).
TestRun	Execution of a set of test cases on a specific version of the test object (after [39]).
TestResult	The consequence/outcome of the execution of a test. It includes outputs to screens, changes to data, reports, and communication messages sent out (after [39]). Normally a test result contains a test verdict that states whether a functional test has passed, failed or none of both. For security testing such a functional test verdict may not be applicable in any case (e.g. when we have found a certain kind of security issue by testing, do we say the test has failed or passed).

These concepts and their relationship is sketched in the following information model.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 123 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

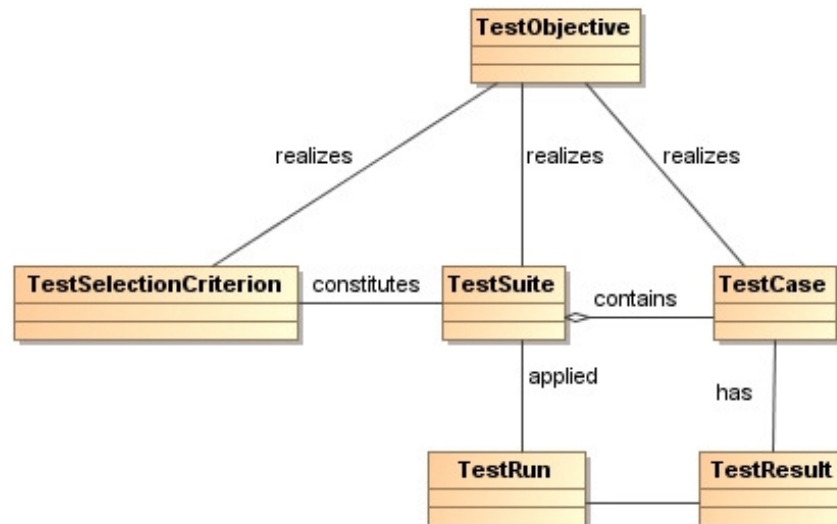


Figure 53: Information model for testing concepts

3.2.4 Risk-Based Security Testing

Risk-based Security Testing can be generally introduced with two different goals in mind. On the one hand side risk based testing approaches can help to optimize the overall test process. The results of the risk analysis, i.e. the results of threat and vulnerability analysis, are used to guide the test identification and may complement requirements engineering results with systematic information concerning threats and vulnerabilities of a system. A comprehensive risk assessment additionally introduces the notion of risk values, that is the estimation of probabilities and consequences for certain threat scenarios. These risk values can be additionally used to weight threat scenarios and thus help identifying which threat scenarios are more relevant and thus identifying the threat scenarios that are the ones that need to be treated and tested more carefully. The basic idea of kind of this kind of risk based security testing approach is more or less similar to other risk based testing strategies already mentioned in the beginning of this chapter (i.e. Section 3.2) and in Deliverable D1.WP2. However, the main challenge remains. Currently there is no method framework that allows for systematically capture security risks (i.e. threat scenarios, vulnerabilities, countermeasures) and risk values and relate both of them to test artefacts so that test identification and test selection is effectively supported.

On the other hand side risk based testing approaches can help to optimize the risk analysis and the risk assessment itself. Risk analysis and risk assessment, similar to other development activities in early project phases, are mainly based on assumption on the system itself. On the other hand testing is one of the most relevant means to do real experiments on a system and thus be able to gain empirical evidence on the existence of vulnerabilities, the applicability and consequences of threat scenarios and the quality of countermeasures. Thus, risk based testing results can be used as a form of evidence for the assumptions that have been made during the risk evaluation and risk assessment. In particular risk based testing may help in

- providing evidence on **functional correctness of countermeasures**,
- providing evidence on the **absence of known vulnerabilities**, and
- discovering **unknown risk factors** (i.e. vulnerabilities).

The results of a risk based testing process then help optimizing risk analysis by identifying new risk factors and reassessing the risk values, that have been stated during system analysis and planning. Figure 54: Risk-based testing shows the overall interaction between risk analysis and testing and depicts both approaches,

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 124 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

the optimization of the testing approach by means of risk analysis results and the control and optimization of the risk analysis and risk assessment results by means of test results.

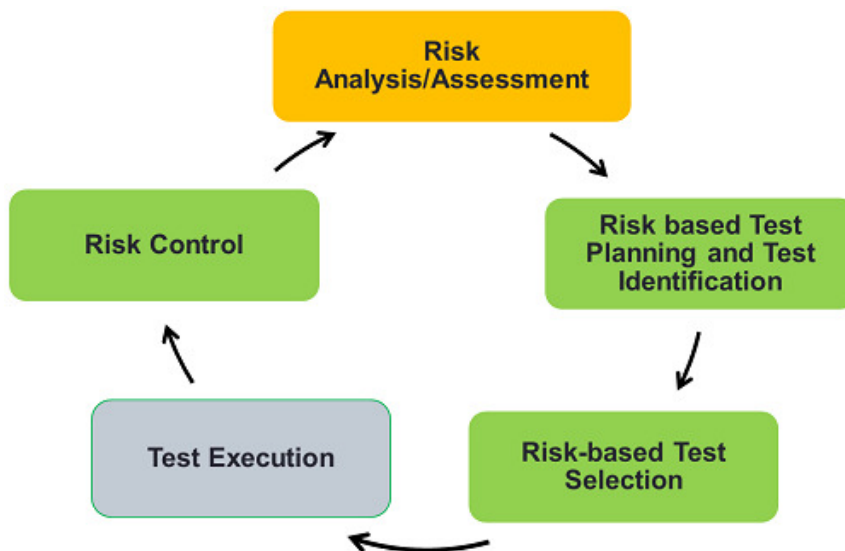


Figure 54: Risk-based testing

In summary we can identify three different activities that can be addressed by risk based testing approaches.

Risk-based test planning: The goal of risk-based test planning is to improve the testing process systematically so that high-risk areas of the application under test can be covered and same time achieve a reduction in the expenses and the resources used by the test work is focused on areas with the highest risks. The risk-based test planning (RBT) are risk factors identified and created test cases according to the system risk.

Risk-based test selection: To find an optimal set of test cases, require an appropriate selection strategy. Such a strategy on the one hand takes into account the available test budget and provides, far as possible, the necessary test coverage on the other hand. In functional testing coverage is often described by the coverage of requirements or the coverage of model elements such as states, transitions or decisions. In risk based testing we aim for the coverage of identified risks of a system. Risk-based test selection criteria can be used to control the selection or the selected generation of test cases. The criteria are designed by taking the risk values from the risk assessment to set priorities for the test case generation as well as to the order of test execution.

Risk control: The decision when a test is to end is always a question of the remaining test-budget, the remaining time and the probability to discover even more critical errors, vulnerabilities or design flaws. In risk based testing risk analysis gives a good guidance where to find critical errors and which kinds of risks have to be addressed (see above). On the other hand, the test results can reflect and verify the assumptions that have been made during risk analysis. Newly discovered flaws or vulnerabilities need to be integrated in the risk analysis. If the tests sufficiently cover the treatments and countermeasures, the number of errors or flaws indicates whether the maturity of treatments and countermeasures is sufficient to provide the required level of protection. In this sense, the test results can be used to adjust risk analysis results by introducing new or revised vulnerabilities or revised risk estimations on basis of the errors or flaws that have been found. Test results, test coverage information and a revised or affirmed risk assessment may provide a solid argument that can be used to effectively verify the level of security of a system.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 125 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

3.2.5 Traceability for Risk-Based Testing

In the following subsection we introduce relations between the concepts from the risk analysis domain and the testing domain. The relations are dedicated to support the risk based testing activities that have been introduced in the section above. To be able to easily identify the source domain for each concept and thus provide a better readability of the overall specification, we prefix each element using a different identifier for each source domain. Thus TEST::TestCase identifies the concept test case from the testing domain and RA::ThreatScenario identifies a threat scenario from the risk analysis domain.

3.2.5.1 Traceability for risk based test planning

To enable risk based test planning we mainly base on the definition of test objectives. Test objectives describe objectives or purposes to be tested. We consider a test objective to be foremost an informal specification that defines what aspect of a certain system, functionality, or protocol etc. should be tested and how these tests can be performed (e.g. which testing methods can be used for testing. Similar to requirements in requirements engineering test purposes can be refined and decomposed during the test development process. In the following we describe the relationship between TEST::TestObjective and RA::UnwantedIncident, RA::ThreatScenario, RA::Vulnerability and RA::TreatmentScenario. While RA::UnwantedIncident, RA::ThreatScenario describe negative requirements on a system with quite similar implications for testing, we consider that a RA::Vulnerability is a potential problem in a system that is to be found or closer characterized by testing and a RA::TreatmentScenario is directly connected to constructive counter measures and thus already relates to system security requirements, which can be tested in a functional manner.

Table 13: Relations between risk assessment concepts and test identification concepts

RA::UnwantedIncident	TEST::TestObjective	Test objectives for an unwanted incident describe what kind of test and test methods can be applied to initiate and detect an unwanted incident and to characterize its consequences. Information on related risks can be used to weight the TestObjective.
RA::ThreatScenario	TEST::TestObjective	Test objectives for threat scenarios describe what kind of test and test methods can be applied to initiate a threat scenario and to characterize its consequences. Information on related risks can be used to weight the TestObjective
RA::Vulnerability	TEST::TestObjective	Test objectives for vulnerabilities describe what kind of test and test methods can be applied to find or characterize a vulnerability. Information on related risks can be used to weight the TestObjective
RA::TreatmentScenario	TEST::TestObjective	Test objectives for treatment scenarios describe what kind of test and test methods can be applied to characterize the maturity and effectiveness of a TreatmentScenario. Information on related risks can be used to weight the TestObjective.

3.2.5.2 Traceability for risk based test selection

The process of test selection is meant to be either accomplished on existing test cases as a preparation of a test run or during test generation to enable a directed or goal oriented generation of tests. The result in both cases is a selection of test cases, i.e. a test suite that conform to a set of selection properties that are given by a test selection criterion. In the case of risk based test selection criteria these properties are defined by means of risk assessment concepts, first and foremost the risks itself.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 126 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

In general the risk weighted test objectives (see Section above) or artefacts that are deduced from risk weighted test artefacts may provide the necessary link that transitively relates test selection criteria to risk assessment artefacts. We would then able to apply a test selection on basis of coverage criteria for risk weighted test objectives. This additionally may provide the opportunity to consider simultaneously properties that come from risk analysis and from other sources. However, this is more a question on optimization and on how we effectively realize traceability. To conceptionally describe the relationship between risk analysis and test selections we going to describe the direct relationship between test concepts and risk assessment concepts for risk based test selection.

Table 14: Relations between risk assessment concepts and test selection concepts

RA::Risk	TEST::TestSelectionCriterion	The test selection criterion directly or indirectly relates to risks (i.e. the product of probability and consequence) and selects test cases with respect to their risk coverage (e.g. which risks are addressed and how often are they are addressed) or risk reduction capabilities .
RA::Threat RA::ThreatScenario RA::TreatmentScenario RA:: UnwantedIncident RA::Vulnerability RA::Asset	TEST::TestSelectionCriterion	The test selection criterion directly or indirectly relates to threats, threat scenarios, treatment scenarios, vulnerabilities and unwanted incidents and selects test cases with respect to their coverage of the given elements (e.g. which elements are addressed by the test case)

3.2.5.3 Traceability for risk control

Risk control deals with the revision of risk assessment results by correcting assumptions on probabilities, consequences or the maturity of treatments scenarios or deals with the completion of risk analysis result by integrating vulnerabilities and thus potentially threats, threat scenarios and unwanted incidents.

Table 15: Relations between risk assessment concepts and testing concepts for risk control

RA::Risk	TEST::TestResult	A Test results relates to the risks, which is addressed/covered by the related test case.
RA::TreatmentScenario	TEST::TestResult	A Test results relates to the treatment scenario, which has been verified or characterized by the related test case.
RA::Threat RA::ThreatScenario RA::UnwantedIncident	TEST::TestResult	A Test results relates to the threat, threat scenario or unwanted incident, which has been realized or characterized by the related test case.
RA::Asset	TEST::TestResult	A Test results relates to the Asset, which risks are addressed/covered by the related test case.
RA::Vulnerability	TEST::TestResult	A Test results relates to the vulnerability, which has been located, characterized or detected by the related test case.

Please note, the relationship between test results and the risk analysis concept is often transitively realized by relating risk assessment concepts to the requirements analysis or the architectural design concepts that have been identified in Section 3.2.2. Thus, first relations between, on the one hand side, requirements, components or interfaces and, on the other hand side, risks assessment concepts like risks, vulnerabilities,

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 127 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

threats etc. is established. Furthermore test cases and test suites (and thus the test results) are as well often related to requirements analysis or the architectural design elements. However, this kind of transitive relationship supports quality statements regarding requirements, components and their interfaces but does not allow to trace between test results and risk analysis because the relations between the transitive partners are often realized with an $n::m$ cardinality.

3.3 TEST CASE SELECTION

Test cases help to determine if a program functions correctly. A big program may need a large number of test cases to cover a reasonable part of its functionality, and sometimes test cases are generated automatically, which may increase the number significantly. Moreover, some test cases may take a relatively long time to run, especially when manual interaction is required. Hence, it is not always possible rerun all test cases every time the source code is changed and the program must be rebuilt.

Test case selection automatically chooses which test cases should be rerun when it is not reasonable to execute them all. Such selection speeds up software development, saves computing resources, and gives faster feedback to software developers. Obviously, test case selection must be faster than running all the test cases. Successful test case selection helps to run the most important test cases and thus assists in reducing the number of software errors and related security risks.

We focus on programs written in the C programming language, although many of the ideas apply to other languages as well. Our test case selection is independent of how test cases are described and executed.

Broadly speaking, this test case selection method consists of three major tasks:

- Record a trace for each passed test case. The trace tells which program elements are required for the test case. The trace contains not only the set of elements but also the order in which the elements are used and how many times they are used. It is assumed that test cases are deterministic so that running the same test case multiple times always produces the same behaviour and trace.
- Analyse the source code change, in order to see which elements have changed. The old and new versions of the program are compared, in a chosen representation and level of granularity.
- Select automatically those test cases for rerunning that use the elements that have changed according to the analysis. Test cases that only use unchanged elements in their trace need not be rerun.

It is possible to analyse code changes and trace test cases in source code level, binary level, and various intermediate levels. Moreover, the level of granularity may range from larger to smaller elements. For example, a straight-forward approach might check which C language source code function implementations have changed. A more elaborate method could also take into account input arguments, in order to analyse the context and use more fine-grained elements.

In this test case selection method, we analyse code in the global Control flow graph (CFG) format. CFG is an intermediate representation of the structure of a program, which among other things can be used for analysing all paths that might be traversed through during execution. In gcc, the CFG is a data structure built on top of an intermediate representation. It is a directed graph where nodes represent basic blocks and edges represent possible transfer of control flow from one basic block to another.

A basic block is a straight sequence of code with one entry point and one exit point. There are no conditional jumps within a basic block. A basic block contains one or more statements, for instance assignment, unconditional function call, return, switch or label statements. CFG edges are links between basic blocks and represent, for example, simple jumps or fall-thru without branching.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 128 of 134
		Version: 1.0
		Date : 26.09.2011
		Status : Final Confid : Public

Test case selection gets information about the program with the help of gcc plug-ins. A plug-in is an optional extension of gcc, a new compilation pass linked to gcc using a shared library. The pass can utilise all the information acquired by earlier passes.

Test case selection uses two gcc plug-ins, trace and CFG:

- Trace plug-in instruments the program to output execution trace at run time. When running test cases, each test case produces one trace file. Traces consist of CFG elements and are thus easy to map to the results of the change analysis.
- CFG plug-in builds a representation of the global CFG of the program, including basic blocks, edges and quite a few other structures, and instruments the program to output the CFG. The CFG does not depend on input arguments or runtime events.

These plug-ins can be applied also to other tasks than test case selection.

3.4 SUMMARY

The introduced combination of risk analysis and security testing shows a high potential to improve the systematic quality assurance of security critical systems. On one hand side risk analysis provides a proper guidance for a systematic test identification and test prioritization. On the other hand side security testing and the analysis of security testing results can provide evidence on assumptions that have been made during risk analysis. The relationship described in Section 3.2 help to understand the dependencies between risk analysis concepts and testing concepts. With sufficient tool support, traceability between risk analysis artifacts and testing artifacts can be operationalized and monitored during the system development. For the next iteration tool support is planned and the approach will be applied systematically to selected case studies.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 129 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

GLOSSARY

Information Security Management System (ISMS)

That part of the overall management system, based on a business risk approach, to establish, implement, operate, monitor, review, maintain and improve information security

Process

Set of interrelated or interacting activities which transforms inputs into outputs

Residual risk

Risk remaining after risk treatment

Risk

Combination of the likelihood of an event and its consequence

Risk analysis

Systematic use of information to identify sources and to estimate the risk.

Risk assessment

Overall process of risk analysis and risk evaluation

Risk avoidance

Decision not to become involved in, or action to withdraw from, a risk situation

Risk criteria

Terms of reference by which the significance of risk is assessed

Risk estimation

Process used to assign values to the probability and consequences of a risk

Risk evaluation

Process of comparing the estimated risk against given risk criteria to determine the significance of the risk

Risk identification

Process to find, list and characterize elements of risk.

Risk management

Coordinated activities to direct and control an organization with regard to risk.

Risk optimization

Process related to a risk to minimize the negative and to maximize the positive consequences and their respective probabilities.

Risk reduction

Actions taken to lessen the probability negative consequences or both, associated with a risk

Risk retention

Acceptance of the burden of loss, or benefit of gain from a particular risk

Risk transfer

Sharing with another party the burden of loss or benefit of gain, for a risk

Threat

A potential source of an incident that may result in adverse changes to an asset, a group of assets or an organization

Vulnerability

Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 130 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

REFERENCES

- [1] S. Abiteboul, R. Hull, V. Vianu, Datalog and Recursion, Addison-Wesley, 1995, Ch. 12, pp. 271–310.
- [2] H. Abdelnur, O. Festor, and R. State, "Kif: A stateful sip fuzzer," in 1st International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm), ACM, Ed., July 2007.
- [3] S. Agrawal, P. P. S. Narayan, J. Ramamirtham, R. Rastogi, M. A. Smith, K. Swanson, M. Thottan: VoIP service quality monitoring using active and passive probes. COMSWARE 2006.
- [4] B.K. Aichernig, H. Brandl, E. Jöbstl and W. Krenn. UML in action: a two layered interpretation for testing. ACM SIGSOFT Software Engineering Notes 2011; 36(1):1–8. Proceedings of UML&FM 2010: Third IEEE International workshop UML and Formal Methods.
- [5] B.K. Aichernig, H. Brandl, E. Jöbstl and W. Krenn. Model-based mutation testing of hybrid systems. Proc. of Formal Methods for Components and Objects (FMCO) 2009, Lecture Notes in Computer Science, vol. 6286, Springer, 2010; 228–249.
- [6] B.K. Aichernig, H. Brandl, E. Jöbstl and W. Krenn. Efficient mutation killers in action. ICST 2011, Fourth International Conference on Software Testing, Verification and Validation, IEEE Computer Society, 2011; 120–129.
- [7] A. Alberts, C. Christopher and A. Dorofee, "OCTAVE Threat Profiles. Software Engineering Institute, Carnegie Mellon University, Criteria Version 2.0", Tech. report CMU/SEI-2001. <http://www.cert.org/archive/pdf/OCTAVETHREATProfiles.pdf>-TR-016. ESC-TR-2001-016, 2001.
- [8] Open Mobile Alliance. Internet Messaging and Presence Service Features and Functions. Approved Version 1.2, January 2005.
- [9] Open Mobile Alliance. Push to Talk over Cellular Requirements. Approved Version 1.0, June 2006.
- [10] D. Angluin (Yale), "Learning regular sets from queries and counterexamples," Information and Computation, 75:87-106, 1987
- [11] K. Apt, M. Van Emden, Contributions to the theory of logic programming, Journal of the ACM (JACM) 29 (3) (1982) 841–862.
- [12] ARP RFC 826. <http://tools.ietf.org/html/rfc826>
- [13] G. Balakrishnan and T. Reps. Analyzing Memory Accesses in x86 Executables. Proc. of "Compiler Construction", 2004.
- [14] G. Balakrishnan and T. Reps. WYSIWYX: What you see is not what you execute. ACM Trans. PLS, 2010
- [15] Banks, Greg and Cova, Marco and Felmetsger, Viktoria and Almeroth, Kevin and Kemmerer, Richard and Vigna, Giovanni. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEer. In: Information Security. Lecture Notes in Computer Science. Katsikas, Sokratis and López, Javier and Backes, Michael and Gritzalis, Stefanos and Preneel, Bart (Eds.), 2006
- [16] S. Becker and H. Abdelnur and R. State and T. Engel. An Autonomic Testing Framework for IPv6 Configuration Protocols. In: Mechanisms for Autonomous Management of Networks and Services. Lecture Notes in Computer Science. Stiller, Burkhard and De Turck, Filip (Eds.). Springer Berlin / Heidelberg. 2010.
- [17] C Bekrar; R. Groz, L. Mounier, "Finding Software Vulnerabilities by Smart Fuzzing," Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on , vol., no., pp.427-430, 21-25 March 2011
doi: 10.1109/ICST.2011.48
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5770635&isnumber=5770588>
- [18] H. Brandl, M. Weiglhofer, B.K. Aichernig. Automated conformance verification of hybrid systems.

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 131 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

International Conference on Quality Software 2010; 0:3–12.

- [19] H. Dai, C. Murphy, G. Kaiser, "Configuration Fuzzing for Software Vulnerability Detection," Availability, Reliability, and Security, 2010. ARES '10 International Conference on , vol., no., pp.525-530, 15-18 Feb. 2010
doi: 10.1109/ARES.2010.22
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5438043&isnumber=5437988>
- [20] J.D. DeMott, R.J. Enbody, and W. F. Punch. "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing." Black Hat USA 2007 & DefCon 15, 2007.
- [21] R. DeMillo, R. Lipton R and F. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer April 1978; 11(4):34–41.
- [22] Top 25 Most Dangerous Software Errors. <http://cwe.mitre.org/top25/>
- [23] ETSI (European Telecommunication Standards Institute): Methods for Testing & Specification (MTS); Model-Based Testing (MBT); Requirements for Modelling Notations, ES 202 951 v 1.1.1, 2011.
- [24] ETSI/ETSI TS 102 165-1 (2009). Telecommunications and Internet converged Services and Protocols for Advanced Networking (TISPAN); Methods and protocols; Part 1: Method and proforma for Threat, Risk, Vulnerability Analysis.
- [25] J-C. Fernandez, L. Mounier und C. Pachon. A Model-Based Approach for Robustness Testing. In: Testing of Communicating Systems. Lecture Notes in Computer Science, 2005, Volume 3502/2005, 313, DOI: 10.1007/11430230_23
- [26] M. Fisher: A Model Checker for Linear Time Temporal Logic. Formal Asp. Comput. (FAC) 4(3):299-319 (1992).
- [27] J.E. Forrester and B.P. Miller. 2000. An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4* (WSS'00), Vol. 4. USENIX Association, Berkeley, CA, USA, 6-6.
- [28] V. Ganesh, T. Leek, M. Rinard, "Taint-based directed whitebox fuzzing," Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on , vol., no., pp.474-484, 16-24 May 2009 ' doi: 10.1109/ICSE.2009.5070546
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5070546&isnumber=5070493>
- [29] S. Gorbunov and A. Rosenbloom. AutoFuzz: Automated Network Protocol Fuzzing Framework. In: IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.8, August 2010. pp. 239-245
- [30] R. Groz, K. Li, A. Petrenko. Verification of Modular Systems with Unknown Components Combining Testing and Inference. Submitted 2011.
- [31] R.G. Hamlet. Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering July 1977; 3(4):279–290.
- [32] P. Herzog: OSSTMM 2.1. Open-Source Security Testing Methodology Manual. Institute for Security and Open Methodologies, 2003
- [33] G. Hoglund and G. McGraw. Exploiting Software: How to Break Code. Addison-Wesly, 2004. ISBN: 978-0-201-78695-8.
- [34] M. Howard and D.E. Leblanc: Writing Secure Code; Microsoft Press, 2002
- [35] Y. Hsu, G. Shu and D. Lee, "A model-based approach to security flaw detection of network protocol implementations," Network Protocols, 2008. ICNP 2008. IEEE International Conference on , vol., no., pp.114-123, 19-22 Oct. 2008
doi: 10.1109/ICNP.2008.4697030
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4697030&isnumber=4697012>

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 132 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

- [36] IEC/FDIS 31010:2009, Risk management — Risk assessment of network data and information Proceedings of SPIE - The techniques, 2009.
- [37] *IEEE 610* – Standard Glossary of Software Engineering Terminology
- [38] ISO/IEC 27000:2009: Information technology — Security techniques — Information security management systems — Overview and vocabulary.
http://standards.iso.org/ittf/PubliclyAvailableStandards/c041933_ISO_IEC_27000_2009.zip
- [39] ISTQB: Standard glossary of terms used in Software Testing, Version 2.1, http://www.german-testing-board.info/downloads/pdf/ISTQB_Glossary_of_Testing_Terms_2-1.pdf
- [40] J. James, L.R. Cebula, A Taxonomy of Operational Cyber Security Risks Carnegie Mellon, Software Engineering Institute, CERT Program, 2010
- [41] C. Jard, T. Jeron. TGV: theory, principles and algorithms. International Journal on Software Tools for Technology Transfer (STTT) 2005; 7(4):297–315.
- [42] A. Jones, A. Jack: An Introduction to Factor Analysis of Information Risk (FAIR); http://www.riskmanagementinsight.com/media/docs/FAIR_introduction.pdf
- [43] J. Jürjens. “Secure Systems Development with UML”. Springer. 2005
- [44] W. Krenn, R. Schlick, B.K. Aichernig. Mapping UML to labeled transition systems for test-case generation – a translation via object-oriented action systems. Proc. of Formal Methods for Components and Objects (FMCO) 2009, Lecture Notes in Computer Science, vol. 6286, Springer, 2009; 186–207.
- [45] F. Lalanne and S. Maag, A Formal Data-centric Approach for Passive Conformance Testing of Communication Protocols. Internal Research Report TSP-11003-LOR., Tech. rep., Telecom SudParis (2011).
- [46] K. Li, R. Groz, K. Hossen, C. Oriat, Inferring Extended Finite State Machine Model Suitable for Security Testing in Internet of Services, (submitted), 2011.
- [47] K. Li, R. Groz, and M. Shahbaz, “Integration Testing of Distributed Components based on Learning Parameterized I/O Models”, In FORTE 2006, LNCS 4229, pp. 436-450, Paris, France, 2006
- [48] M. S. Lund. Operational analysis of sequence diagram specifications. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2008.
- [49] M. S. Lund and B. Solhaug and K. Stølen. Model-Driven Risk Analysis. The CORAS Approach. 1st Edition., 2011, XVI, 400 p., Hardcover. ISBN: 978-3-642-12322-1 (To appear).
- [50] Y.S. Ma, J. Offutt, Y.R. Kwon. MuJava: a mutation system for Java. ICSE '06: Proceedings of the 28th international conference on Software engineering, ACM, 2006; 827–830.
- [51] T. Masse, S. O’Neil and J. Rollins. The Department of Homeland Security’s Risk Assessment Methodology: Evolution, Issues, and Options for Congress *The Department of Homeland Security’s Risk Assessment Methodology*, 2007.
- [52] S. Mauw. and M. Oostdijk: Foundations of Attack Trees. Information Security and Cryptology - ICISC 2005, Springer Berlin / Heidelberg, 2006, 3935, 186-198
- [53] C.C. Michael and W. Radosevich: Risk-Based and Functional Security Testing; Cigital, Inc., 2005
- [54] B. Miller, L. Fredriksen, B. So; “An Empirical Study of the Reliability of UNIX Utilities”, Proceedings of the 4th USENIX Windows System Symposium, 2000, pp.59-68
- [55] B. P. Miller, G. Cooksey, and F. Moore. 2007. “An empirical study of the robustness of MacOS applications using random testing.” SIGOPS Oper. Syst. Rev. 41, 1 (January 2007), 78-86. DOI=10.1145/1228291.1228308 <http://doi.acm.org/10.1145/1228291.1228308>
- [56] G. Morales, S. Maag, A. Cavalli, W. Mallouli, E. Montes de Oca, and B. Wehbi. Timed Extended

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 133 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

- Invariants for the Passive Testing of Web Services . The 8th International Conference on Web Services (ICWS 2010), Miami, Florida, USA, July 5-10, 2010.
- [57] K.K. Murthy, K.R. Thakkar and S. Laxminarayan: Leveraging Risk Based Testing in Enterprise Systems Security Validation. Proc. First Int Emerging Network Intelligence Conf, 2009, 111-116.
 - [58] D.S. Nielsen, The Cause/Consequence Diagram Method as a Basis for Quantitative Accident Analysis: Danish Atomic Energy Commission, RISO-M-1374, 1971.
 - [59] U. Nilsson, J. Maluszynski, Logic, programming and Prolog, 2nd Edition, Vol. 5, Wiley, 1990.
 - [60] S. Nürnberger, R. Karnapke, J. Nolte: Sensorium - An Active Monitoring System for Neighborhood Relations in Wireless Sensor Networks. ADHOCNETS 2010:34-47
 - [61] P. Oehlert, "Violating assumptions with fuzzing," Security & Privacy, IEEE , vol.3, no.2, pp. 58- 62, March-April 2005 doi: 10.1109/MSP.2005.55
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1423963&isnumber=30742>
 - [62] OMG Unified Modeling Language (OMG UML), Superstructure v2.3, formal/2010-05-05. OMG specification, OMG (May 2005)
 - [63] T. Ormandy (Google), Making software dumber. 2010.
http://taviso.decsystem.org/making_software_dumber.pdf
 - [64] S. Rawat and L. Mounier, Offset-Aware Mutation based Fuzzing for Buffer Overflow Vulnerabilities: Few Preliminary Results, In Second International Workshop on Security Testing (SECTEST 2011), Workshop of the IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST2011), 2011
 - [65] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261. Technical report, Internet Engineering Task Force, 2002.
 - [66] P. Saitta, B. Larcom and M. Eddington: Trike v.1 Methodology Document; 2005
 - [67] H. Schmidt and J. Jürjens. UMLsec4UML2 – Adopting UMLsec to Support UML2. Technical Report No 838; Technische Universität Dortmund; 2011.
 - [68] S. Spark, S. Embleton, R. Cunningham and C. Zou, "Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting," Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual , vol., no., pp.477-486, 10-14 Dec. 2007
doi: 10.1109/ACSAC.2007.27
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4413013&isnumber=4412960>
 - [69] V. Stolz, Temporal Assertions with Parametrized Propositions, Journal of Logic and Computation 20 (3) (2008) 743–757.
 - [70] K. Takahisa, H. Miyuki and K. Kenji, "AspFuzz: A state-aware protocol fuzzer based on application-layer protocols," Computers and Communications (ISCC), 2010 IEEE Symposium on , vol., no., pp.202-208, 22-25 June 2010
doi: 10.1109/ISCC.2010.5546704
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5546704&isnumber=5546495>
 - [71] A. Takanen, Diamonds deliverable D1.WP1: "Use Case Overviews and Requirements."
 - [72] A. Takanen: "Fuzzing: the Past, the Present and the Future". Actes du 7ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC), 2009. pp. 202-212
 - [73] A. Takanen, J. DeMott and C. Miller "Fuzzing for software security testing and quality assurance". Artech House. 2008.
 - [74] A. Takanen, J. DeMott and C. Miller, "Software Security Assessment through Specification Mutations and Fault Injection". In: Communications and Multimedia Security Issues of the New Century, Series:

	Concepts for Model-Based Security Testing Deliverable ID: D2.WP2	Page : 134 of 134
		Version: 1.0 Date : 26.09.2011
		Status : Final Confid : Public

- IFIP Advances in Information and Communication Technology, Vol. 64, Steinmetz, Ralf; Dittmann, Jana; Steinebach, Martin (Eds.), 2001
- [75] G. Tian-yang, S. Yin-sheng and F. Yuan: Research on Software Security Testing World Academy of Science, Engineering and Technology 69 2010, 2010
 - [76] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools* 1996; 17(3):103–120.
 - [77] I. Uusitalo, Diamonds deliverable D1.WP2: “Review of Security Testing Tools”.
 - [78] I. Uusitalo, Diamonds deliverable D2.WP3: “Initial Design of Security Testing Tools.”
 - [79] M. Utting and B. Legeard, *Practical Model-Based Testing – A Tools Approach*, Morgan&Kauffmann, 2007.
 - [80] M. Van Emden, R. Kowalski, The semantics of predicate logic as a programming language, *Journal of the ACM (JACM)* 23 (4) (1976) 733–742.
 - [81] J. Viide, A. Helin, M. Laakso, P. Pietikäinen, M. Seppänen, K. Halunen, R. Puuperä, and J. Röning. 2008. Experiences with model inference assisted fuzzing. In *Proceedings of the 2nd conference on USENIX Workshop on offensive technologies (WOOT'08)*. USENIX Association, Berkeley, CA, USA, Article 2 , 6 pages.
 - [82] Y. Yang, H. Zhang, M. Pan, J. Yang, F. He and Z. Li, "A Model-Based Fuzz Framework to the Security Testing of TCG Software Stack Implementations," *Multimedia Information Networking and Security*, 2009. MINES '09. International Conference on , vol.1, no., pp.149-152, 18-20 Nov. 2009
doi: 10.1109/MINES.2009.111
URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5368443&isnumber=5368002>
 - [83] X. Zhu, Z. Wu, J. Atwood. A New Fuzzing Method Using Multi Data Samples Combination. *Journal of Computers, North America*, 6, may. 2011. Available at: <<http://ojs.academypublisher.com/index.php/-jcp/article/view/0605881888>>. Date accessed: 11 Oct. 2011.
 - [84] L. Lamport. How to write a long formula. *Formal Aspects of Computing*, 6(5):580–584, 1994.
 - [85] F. Seehusen, B. Solhaug, and K. Stølen. Adherence preserving refinement of trace-set properties in STAIRS: exemplified for information flow properties and policies. *Journal of Software and Systems Modeling*, 8(1):45–65, 2009.