



Title: Final Security Testing Techniques

Version: 1.0
Date : 23.05.2013
Pages : 80

Author: Stephane Maag

Reviewers: VTT, INP, Codenomicon

To: DIAMONDS Consortium

The DIAMONDS Consortium consists of:

Codenomicon, Conformiq, Dornier Consulting, Ericsson, Fraunhofer FOKUS, FSCOM, Gemalto, Get IT, Giesecke & Devrient, Grenoble INP,itrust, Metso, Montimage, Norse Solutions, SINTEF, Smartesting, Secure Business Applications, Testing Technologies, Thales, TU Graz, University Oulu, VTT

Status:

[] Draft
[] To be reviewed
[] Proposal
[X] Final / Released

Confidentiality:

[X] Public Intended for public use
[] Restricted Intended for DIAMONDS consortium only
[] Confidential Intended for individual partner only

Deliverable ID: D5.WP2

Title: Final Security Testing Techniques

Summary / Contents:

This document describes the security-testing techniques defined by the DIAMONDS consortium and used through the different WP1 real use cases.


Contributors to the document:

Josip Bozic (TUGraz), Franz Wotawa (TUGraz), Bruno Legeard (SMT), Julien Botela (SMT), Pramila Mouttappa (IT), Stephane Maag (IT), Anderson Morais (IT), Kati Kittila (Codenomicon), Ari Takanen (Codenomicon), Sanjay Rawat (Grenoble INP), Fabien Duchène (Grenoble INP), Jean-Luc Richier (Grenoble INP), Wissam Mallouli (Montimage), Fredrik Seehusen (SINTEF), Martin Schneider (FOKUS), Johannes Viehmann (FOKUS), Rautila Mika (VTT), Sami Nojonen (VTT), Alex Mckinnon (itrust), Carlo Harpes (itrust)

TABLE OF CONTENTS

1. Introduction	8
2. Monitoring and inspection techniques	9
2.1 Events based monitoring for security checking.....	9
2.1.1 Description.....	9
2.1.2 Innovation with respect to the STOA	11
2.1.3 Applications and contributions to case studies.....	12
2.1.4 Metrics and results	12
2.1.5 Tools support.....	12
2.1.6 Exploitation and dissemination	13
2.2 Symbolic Passive Testing	14
2.2.1 Description.....	14
2.2.2 Innovation with respect to the STOA	19
2.2.3 Applications and contributions to case studies.....	19
2.2.4 Metrics and results	19
2.2.5 Tools support.....	20
2.2.6 Exploitation and dissemination	20
2.3 Distributed and Cooperative Intrusion Detection.....	20
2.3.1 Description.....	20
2.3.2 Innovation with respect to the STOA	22
2.3.3 Applications and contributions to case studies.....	22
2.3.4 Metrics and results	22
2.3.5 Tools support.....	23
2.3.6 Exploitation and dissemination	23
2.4 Anomaly Detection with Machine Learning Approach in ICS Network Monitoring.....	23
2.4.1 Description.....	23
2.4.2 Innovation with respect to the STOA	24
2.4.3 Applications and contributions to case studies.....	24
2.4.4 Metrics and results	24
2.4.5 Exploitation and dissemination	24
2.5 Passive testing and network monitoring.....	24
2.5.1 Description.....	24
2.5.2 Innovation with respect to the STOA	25
2.5.3 Applications and contributions to case studies.....	26
2.5.4 Metrics and results	26
2.5.5 Tools support.....	26
2.5.6 Exploitation and dissemination	26
3. Active Security Testing Techniques	27
3.1 Attack Pattern Based Testing.....	27
3.1.1 Description.....	27
3.1.2 Innovation with respect to the STOA	27
3.1.3 Applications and contributions to case studies.....	28
3.1.4 Metrics and results	31
3.1.5 Tool support.....	32
3.1.6 Exploitation and dissemination	33
3.2 Model-based security testing from security test purposes and behavioral models.....	33
3.2.1 Description.....	33
3.2.2 Innovation with respect to the STOA	34

3.2.3	Tools Support	35
3.2.4	Applications and contributions to case studies	35
3.3	Model Inference Assisted Evolutionary Fuzzing	35
3.3.1	Description.....	35
3.3.2	Innovation with respect to the STOA	39
3.3.3	Applications and contributions to case studies.....	39
3.3.4	Metrics and results	39
3.3.5	Tool support.....	39
3.3.6	Exploitation and dissemination	39
3.4	Static Taintflow Analysis for Vulnerability Detection in Binary Code.....	40
3.4.1	Technique Description	40
3.4.1.3	<i>Interprocedural Dataflow Analysis (InterDF)</i>	42
3.4.2	Innovation with respect to the STOA	43
3.4.3	Applications and contributions to case studies.....	43
3.4.4	Metrics and results	43
3.4.5	Tool support.....	43
3.4.6	Exploitation and dissemination	43
3.5	Active Fuzz Testing.....	44
3.5.1	Description.....	44
	Fuzz test generation strategies	53
3.5.2	Innovation with respect to the STOA	55
3.5.3	Applications and contributions to case studies.....	56
3.5.4	Metrics and results	56
3.5.5	Tool support.....	56
3.5.6	Exploitation and dissemination	56
3.6	Model-Based Behavioural Fuzzing	56
3.6.1	Technique description	57
3.6.2	Innovation with respect to the STOA	60
3.6.3	Applications and contributions to case studies.....	60
3.6.4	Metrics and results	60
3.6.5	Tool support.....	61
3.6.6	Exploitation and dissemination	61
3.7	Source code change based test case selection.....	61
3.7.1	Technique description	61
3.7.2	Innovation with respect to the STOA	62
3.7.3	Applications and contributions to case studies.....	62
3.7.4	Metrics and results	64
3.7.5	Tool support.....	64
3.7.6	Exploitation and dissemination	64
3.8	Fault detection.....	64
3.8.1	Fault detection technique description	64
3.8.2	Innovation with respect to the STOA	65
3.8.3	Applications and contributions to case studies.....	65
3.8.4	Metrics and results	65
3.8.5	Tool support.....	65
3.8.6	Exploitation and dissemination	65
3.9	OWASP-based web security testing	65
3.9.1	Technique description	65
3.9.2	Innovation with respect to the STOA	67
3.9.3	Applications and contributions to case studies.....	67

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 4 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public


3.9.4	Metrics and results	67
3.9.5	Tool support.....	67
3.9.6	Exploitation and dissemination	68
4.	Risk Analysis for Risk Based Testing	69
4.1	Risk-based test identification and prioritization	69
4.1.1	Technique description	69
4.1.2	Innovation with respect to the STOA	71
4.1.3	Applications and contributions to case studies	72
4.1.4	Metrics and results	72
4.1.5	Tool support.....	73
4.1.6	Exploitation and dissemination	73
4.2	Risk Assessment Based on Vulnerability Analysis	73
4.2.1	Description.....	73
4.2.2	Innovation with respect to the STOA	74
4.3	Compositional Risk Assessment.....	76
4.3.1	Technique description	76
4.3.2	Innovation with respect to the STOA	76
4.3.3	Applications and contributions to case studies	76
4.3.4	Metrics and results	77
4.3.5	Tool support.....	77
4.3.6	Exploitation and dissemination	77
References	78

FIGURES

Figure 1: MMT global architecture.....	13
Figure 2: Sequence diagram – Session establishment and Bluestabbing attack	15
Figure 3: IOSTS model – Session establishment and Bluestabbing attack	16
Figure 4: Symbolic execution of IOSTS	17
Figure 5: Distributed intrusion detection architecture	20
Figure 6: SQLI Attack model	28
Figure 7: Transition between states	29
Figure 8: GUI of the SUT.....	29
Figure 9: Attack pattern model for DVWA	31
Figure 10: Smartesting process and tool for model-based security testing	34
Figure 11: Overview of the proposed approach, showing the main three components	35
Figure 12: Pseudo code of Genetic Algorithm.....	37
Figure 13: Input Grammar to generate fuzzed inputs (excerpt)	38
Figure 14: Call graph slice.....	40
Figure 15: Pseudo-code of the Technique	42
Figure 16: The three different approaches to testing	44
Figure 17: Positive vs. negative testing.....	45
Figure 18: Specified vs. implemented functionality	45
Figure 19: The cost of a bug depending on when it is discovered	46
Figure 20: Attack vectors and surfaces	46
Figure 21: Types of anomalies Defensics produces, and some types of problems it is able to find	47
Figure 22: Editor for sequence models and message structures	53
Figure 23: Web server example for fuzzing.....	54
Figure 24: Application of a behavioural fuzzing operator in order to generate an invalid sequence	57
Figure 25: Sequence diagram with extended RBAC stereotype	59
Figure 26: Python based unit testing environment.....	63
Figure 27: Makefile rules	63

TABLES

Table 1: Slice table for a sample Bluetooth trace τ	18
Table 2: Evaluation table for each trace slice.....	18
Table 3: Prototype tool results on sample Bluetooth traces	19
Table 4: Execution time for DVWA, Mutillidae and Bodgelt.	32
Table 5: CVSS Vectors and CVSS Base Scores for IMS telecommunication core network	75


	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 6 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

HISTORY

Vers.	Date	Author	Description
0.1	2013/3/13	Stephane Maag	Document creation
0.8	2013/5/8	Stephane Maag	Refactoring the document
1.0	2013/5/23	Pramila Mouttappa	Updation of final review comments

APPLICABLE DOCUMENT LIST


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

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 7 of 80
		Version: 1.0 Date : 23.05.2013
		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 reviewed the state-of-the-arts methods used in security testing. D2.WP2 described the main concepts dedicated to model-based security testing used by the different project partners. In D3.WP2 more concrete model-based security testing methods were discussed and introduced. In this document D5.WP2 we describe the final security testing techniques by the different project partners. This deliverable consists of three parts: Monitoring and inspection, active testing and risk analysis for risk based testing.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 8 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

1. INTRODUCTION

Testing security properties such as confidentiality, integrity, authentication, authorization, availability, and non-repudiation is of high importance to ensure trust in the behaviour of the systems. In general and as previously mentioned in our previous WP2 deliverables, security testing activities are divided into functional security testing and security vulnerability testing. When beginning the DIAMONDS project, systematically testing a system for its security was a relatively new concern. Based on that, three main objectives were defined: (i) the design of new techniques for the modelling of security properties, test generation and test execution, (ii) the adaptation of existing tools provided by academic and industrial partners for security testing, and (iii) the introduction of risk and failure analysis in the test process. Through this deliverable and the works performed in this work package, we bring novel solutions and demonstrate that all targeted purposes were successfully reached.


Several complementary approaches are illustrated herein. Furthermore, in order to ease the reading and to help focusing on the specific goal when finding out a solution to a security testing issues, all below mentioned sections are detailed according to the same structure. Indeed, for each approach, the reader will have an overview regarding: (i) its description, (ii) its innovation w.r.t the state of arts, (iii) how it has been applied to the WP1 case studies, (iv) the metrics and results, (v) the eventual tool support and (vi) and exploitation and dissemination aspects illustrating the quality of the proposed techniques.

In this document, the final model-based security testing techniques defined, developed and used by the DIAMONDS partners through different industrial case studies are documented. This report is organized following the three main types of approaches: monitoring, active and risk based testing.

In Chapter 1 we present the monitoring, inspection and passive testing approaches. Four different and complementary techniques are depicted. They are based on formal methods and mainly dedicated to DPI, network and services monitoring. Besides, multi data sources management systems as well as distributed network intrusion detection approaches are performed. All these techniques have been applied to different WP1 use cases and are now also transferred through industrial tools or prototypes.

Chapter 2 deals with active testing techniques. While discussing the overall active security testing methodologies, we describe foundations for vulnerability detection based on model-based, behavioural and fuzz testing. Several different approaches are detailed based on many complementary techniques. Attack patterns, vulnerability properties such as fuzzing are proposed to tackle same purposes.

Finally, the last Chapter proposes risk analysis for testing based on risk models and vulnerability analysis. Risk assessment and test identification and prioritization are demonstrated mainly to tackle security assessment.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 9 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

2. MONITORING AND INSPECTION TECHNIQUES

2.1 EVENTS BASED MONITORING FOR SECURITY CHECKING

2.1.1 Description

2.1.1.1 Motivations

New and more critical vulnerabilities are constantly being introduced by the evolution of the Internet and mobile communications where critical infrastructures are more and more open and corporate IT is more and more dematerialized (e.g., using cloud services). This is pushing towards the need for more proactive and automated mechanisms for detecting and preventing anomalies due to attacks or misbehaviours.

In this context, Deep Packet Inspection (DPI) is considered as a key element in the shift towards advanced monitoring. DPI is the process of capturing network traffic, analysing and inspecting it in detail to determine accurately what is really happening in the network. This “core component” feeds the different monitoring applications with high added value information.

Starting from this perception, the requirements of a network monitoring system can be summarized as follows:

- High capturing performance. It must be able to capture traffic at high speeds and under high traffic volume. This depends on what is to be monitored and where.
- Extensibility. If new services are integrated in the network, it must be possible to deploy effortlessly new monitoring mechanisms for these specific services. In addition, if new analysis techniques are needed, it should be possible to integrate them as effortlessly as possible.
- Scalability. It must be able to handle the increase of traffic data as network link speeds and the number of probes increase in the network without performance degradation. Scalability can be achieved by reducing the traffic information collected using efficient packet capturing mechanisms and traffic pre-processing.
- Real time functioning. It must implement real time mechanisms in order to quickly detect network security/performance problems and allow timely execution of automated or manual countermeasures.
- Granularity. It must be able to track the security and performance of each service by capturing and analysing the traffic belonging to the application of interest.
- Diversity. It has to support the network's diversity as it contains different types of network devices from multiple vendors, protocols stacks, and applications to provide services to the user.
- Low cost. It should not use excessive amount of computing, storage, and communication resources so the cost of deploying and operating the monitoring infrastructure is low for service providers.
- Secure. It should not add vulnerabilities to the network, or disturb normal network operation.


2.1.1.2 Events based monitoring technique

In this section, we present Montimage monitoring technique that allows providing a real-time visibility of network traffic. It provides network, application, flow and user level visibility. This technique facilitates network security and performance monitoring and operation troubleshooting. It is based on a rules engine can correlate network and application events in order to detect operational, security and performance incidents. The security properties (called MMT¹-Security properties) within this engine intend to formally specify security goals and attack behaviours related to the application or protocol under test. Their model is inspired from LTL logic² and can refer to two types of properties: “Security rules” and “Attacks” described as follows:

- A Security rule describes the expected behaviour of the application or protocol under-test whether it is functional or security oriented. The non-respect of the security property indicates an abnormal behaviour, e.g. the access to a specific service must always be preceded by an authentication phase.

¹ MMT refers to Montimage Monitoring Tool that implements the current technique.

² Alessandro Armando, Roberto Carbone, and Luca Compagna. LTL model checking for security protocols. Journal of Applied Non-Classical Logics, 19(4):403–429, 2009.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 10 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

- An Attack describes a malicious behaviour whether it is an attack model, a vulnerability or a misbehaviour. Here, the respect of the security property indicates the detection of an abnormal behaviour 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 behavioural attack.

It must be noted that the events that we take into account 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. The main definition of an MMT-Security property is provided by definition number 1. The other definitions allow understanding the basics of the used model.

Definition 1 (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 (also called security rule) 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_j , $j \in \{1, \dots, m\}$) 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.

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.
- 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.

2.1.1.3 Multi data sources management for security analysis

In the context of events based monitoring technique, DPI (Deep Packet Inspection) and DFI (Deep Flow Inspection) 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 data levels, to detect 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:

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 11 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

- *Real attributes:* They can be directly extracted from the inspected packet. They correspond to a protocol field value.
- *Calculated 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 for the security analysis engine.
- *Meta attributes:* These attributes are linked to each packet to describe capture information. The time of capture of each packet (timestamp attribute) is the main meta-attribute in the current version of MMT.

The extracted attributes needed for security analysis can emanate from different data sources (probes and/or interfaces). This is managed in the security analysis solution during the specification phase of the security properties. Indeed, the data sources identifiers are part of the meta-attributes that can be used in the specification of the relevant events for security analysis. Three architectures are taken into account in MMT:

- *Local analysis:* the collected traffic is analysed for security purposes in one probe that captures network traffic from one or several interfaces.
- *Centralized analysis:* the traffic capture is distributed but the security analysis is centralized. All data sources send their collected traffic (filtered or not) to the same master server that correlates the traces (i.e., need to synchronize probes to be able to perform this task).
- *Distributed analysis:* the traffic capture is distributed and the analysis is performed by all the probes that communicate together to share information. This analysis can be very interesting in some specific case studies like ad hoc networks.

2.1.2 Innovation with respect to the STOA

The originality of the MMT security properties with respect to existing intrusion detection techniques lies in that they are not based on just pattern matching (i.e., signatures) as in SNORT³ nor requiring writing executable scripts as in BRO⁴. They allow a more abstract description of sequence of events that can represent normal/abnormal behaviour. They can also integrate pattern matching, statistics and machine learning techniques.

- Use of formal security properties to describe both wanted and unwanted behaviour.
- Not exclusively based on pattern matching like most intrusion detection techniques.
- More abstract description of sequence of events (MMT properties).
- Can integrate performance indicators, statistics and machine learning techniques.
- Can integrate countermeasures (e.g., using the iptables library).
- Manages events from different data sources (interfaces or probes).

The technique can be applied to several domains (at protocol, application and business levels) since it is based on a plugin architecture that can parse any structured data and retrieve relevant information for security analysis.

One major innovation of this passive testing technique is its **complementarity** with active testing approaches. Regarding the software development life cycle, active testing is usually applied in the testing phase and conformance testing is already a standard (ISO 9646. Information Technology, Open Systems Interconnection, "Conformance Testing Methodology and Framework". International Standard IS-9646. ISO, 1991. CCITT X.290–X.294). Whereas, passive testing is usually applied in the operation and maintenance phase. In the literature, integration of active and passive testing is rarely elaborated⁵. Nevertheless, it can be very interesting to spend less effort (time and money) to build the behavioural model during the modelling activity.

³ <http://www.snort.org/>

⁴ <http://www.bro.org/>

⁵ Yixin Zhao, Jianping Wu, Xia Yin: From Active to Passive - Progress in Testing Internet Routing Protocols. J. Comput. Sci. Technol. (JCST) 17(3):264-283 (2002)

Indeed, the observations (desired behaviour) are not specified or partially specified within the functional model and the stimuli generation can be random or based on intelligent fuzzing. Besides, the test purposes that guide the test cases generation are not specifically related to the security properties that we want to check on the collected traces (data sources can be distributed). Finally, the integration of the two techniques is very interesting when the tester has a partial knowledge of the impact of its stimuli on the SUT, for example, an attack can violate more than one security property.

2.1.3 Applications and contributions to case studies

The events based monitoring solution has been applied to three case studies:

- Radio protocols case study provided by Thales Communication & Security
- Smartcards and the mobile NFC ecosystem case study provided by Gemalto
- Automotive case study provided by Dornier Consulting

2.1.4 Metrics and results

Case study	Offline/Online	Number of designed properties	Number of analysed traces (including traffic or server logs)	Results
Radio protocols case study	Both	19	17	All attacks implemented and applied are detected
Radio protocols case study	Both	19	17	All attacks implemented and applied are detected
Smartcards and the mobile NFC ecosystem case study	Offline	9	3	No security flaws detected
Automotive case study	Offline	3	2	No security flows detected

More details are presented in D5.WP1 deliverable.

2.1.5 Tools support

The events based monitoring technique has been implemented in MMT tool that stands for Montimage Monitoring Tool. This tool is composed of three complementary, but independent, modules:

- *MMT-Extract* is the core packet processing module. It is a C library that analyses network traffic using Deep Packet/Flow Inspection (DPI/DFI) techniques in order to identify network and application based events by analysing: protocols' field values; network and application Quality of Service (QoS) parameters; and, Key Performance Indicators (KPI). In a similar way, it also allows analysing any structured information generated by applications (e.g., traces, logged packets). MMT-Extract incorporates a plugin architecture for the addition of new protocols or packets, and a public API for integration into third party probes.
- *MMT-Security* is a security analysis engine based on MMT-Security properties. MMT-Security analyses and correlates network and application events to detect operational and security incidents. For each occurrence of a security property, MMT-Security allows detecting whether it was respected or violated.
- *MMT-Operator* is a visualization application for MMT-Security currently under development. It allows collecting and aggregating security incidents to present them via a graphical user interface. MMT-

Operator is conceived to be customizable, i.e., the user will be able to define new views or customize one from a large list of predefined views. With its generic connector, MMT-Operator can be integrated with third party traffic probes.

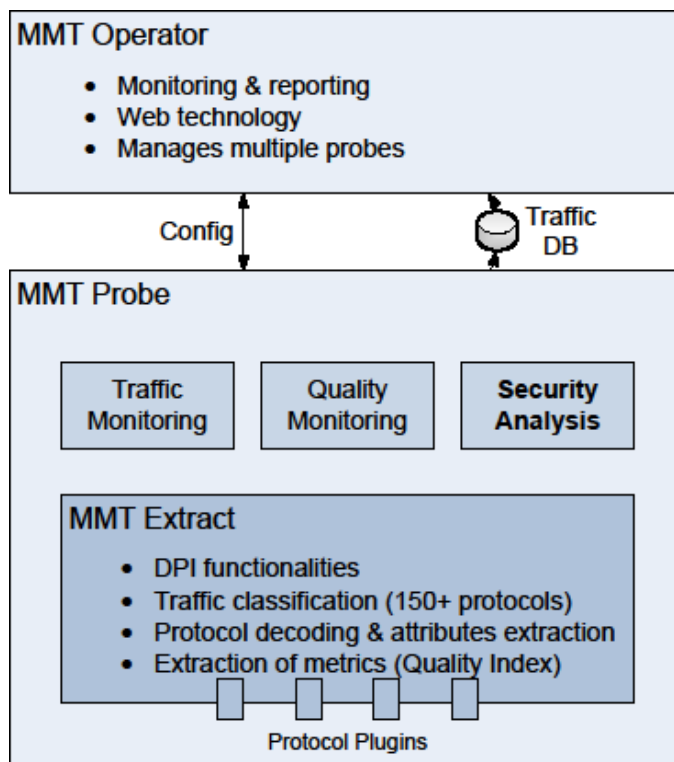


Figure 1: MMT global architecture.


More details of MMT tool are provided in D5.WP3 deliverable.

2.1.6 Exploitation and dissemination

In the context of the DIAMONDS project, the main exploitation objective of Montimage is to industrialize its monitoring solution: MMT (Montimage Monitoring Tool). Several fast exploitations activities have been performed during the Diamonds project life:

What	How	Where	When	Current status
Security analysis module	Updated product	Commercially available	2012	Released
Performance analysis module	Updated product	Commercially available	2012	Released
Two industrial contracts	Adaptation of MMT tool	France	2012	Released
Open source project	ore part of Montimage Monitoring Tool	available under GPL license	2013	Released

The events-based solution have also disseminated in several conferences and workshops (e.g. Sectest 2012, STV 2012). Several demonstrations of the tool have been done in forums and summer schools (Tarot 2011 and Tarot 2012). More details are provided in D5.WP5 deliverable.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 14 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

2.2 SYMBOLIC PASSIVE TESTING

Symbolic passive testing of a property on a real execution trace integrates two important techniques: symbolic execution of an Input-Output Symbolic Transition Systems and a parametric trace slicing approach. Input-Output Symbolic Transition Systems (IOSTS) are commonly used for formally modelling communicating systems interacting with their environment. In IOSTS, the parameters and variable values are represented by symbolic values (called fresh variables) instead of concrete ones. Enumeration of data values is therefore not required. This allows reducing the huge amount of data values commonly applied in many passive testing approaches. We use this formalism to specify the protocol properties as well as several kinds of attack patterns. This helps to detect conformance as well as security anomalies.

Besides, a Parametric trace slicing technique [26] is used for trace analysis Trace analysis plays a very important part in passive testing. A parametric trace is defined as a trace containing events with parameters that have been bound to a concrete data value (i.e., valuation) and parametric trace slicing is defined as a technique to slice (or cut) the real protocol execution trace into various slices based on the valuation. Each slice corresponds to a particular valuation. These trace slices merged together constitutes the execution trace. We then apply the symbolic execution of our properties on the trace slices to provide a test verdict. In order to exemplify our approach, we apply our methodology on the automotive use case for monitoring the Bluetooth protocol behaviour as well as certain attack patterns.

2.2.1 Description

2.2.1.1 INPUT-OUTPUT SYMBOLIC TRANSITION SYSTEMS (IOSTS)

2.2.1.1.1 Syntax of IOSTS

Definition 1. (IOSTS) An IOSTS [27] \mathcal{M}_I is a tuple $\langle D, I, L, l^0, \Sigma, T \rangle$ where:

- $D = V \cup P$ is a finite set of *typed data* which consists of set V of *variables* and set P of *parameters*.
- I is the *initial condition*, a boolean expression on V .
- L is a non-empty, finite *set of locations*.
- $l^0 \in L$ is the *initial location*.
- $\Sigma = \Sigma^? \cup \Sigma^!$ is a non-empty, finite *alphabet of actions* which consists of two mutually disjoint alphabets of *output actions* $\Sigma^!$ and *input actions* $\Sigma^?$, i.e. $(\Sigma^? \cap \Sigma^! = \emptyset)$. In order to distinguish an input from an output action, we may respectively attach the '?' and '!' symbols to the actions.
- T is a finite set of *symbolic transitions*. Each symbolic transition is a tuple $t = \langle l, a, G, A, l' \rangle \in T$ consisting of:
 - a *location* $l \in L$, the *origin* of the symbolic transition.
 - an *action* $a \in \Sigma$ called the *action* of the transition.
 - a *predicate* G on $V \cup P$, called the *guard* is a Boolean expression containing the truth values *true*, *false*.
 - a set of *assignment* A , each assignment is of the form $(x := A_x)_{x \in V \cup P}$, such that, for each $x \in V \cup P$, the right-hand side A_x of the assignment is an expression on $V \cup P$. These assignments are well-typed, that is, the expressions A_x returns a data type which is the same as that of x .
- a *location* $l' \in L$, the *destination* of the transition.

2.2.1.1.2 Semantics of an IOSTS

In general, valuation means assigning concrete values to V or P . In this approach, the concrete input values and initialization values of variables are replaced by symbolic ones, called *fresh variables*. We represent the set of fresh variables by F , where $F \cap V = \emptyset$.

The semantics of an IOSTS $\langle D, I, L, l^0, \Sigma, T \rangle$ is an Input-Output Labelled Transitions Systems (IOLTS) $\langle S, S_0, A, \rightarrow \rangle$ defined as:

- the set of states is $S = L \times \mathcal{V}$, where \mathcal{V} is the set of valuation for the variables V . Formally a state is a pair $\langle l, v \rangle$ where $l \in L$ is a location and $v \in (\mathcal{V}_x)_x \in V$.

- the set of initial states is $S_0 = l^0 \times v$, an initial state is a state $\langle l^0, v^0 \rangle$ such that $l^0 \in L$ is the initial location and v^0 is the valuation of the variables that satisfies the initial condition I , i.e. $\{ S_0 = \langle l^0, v^0 \rangle \mid I(v^0) = true \}$ and $v, v^0 \in \mathcal{V}$.

- the set of valued actions A . The set of valued parameters P is given by Π .
- \rightarrow is the transition relation, which is a 3-tuple $\langle s, \alpha, s' \rangle$ where, $s = \langle l, v \rangle, s' = \langle l', v' \rangle$ are the source and destination states respectively and $\alpha = \langle a, v \rangle$ is a valued action, where $a \in \Sigma$ is the action of the symbolic transition t and v is a valuation of the parameters carried by the action a .

Example of a Bluetooth behaviour defined as an IOSTS

In our approach, we consider only the host controller interface (HCI) layer messages (collected from one device) to describe the device discovery and connectivity behaviour between the Bluetooth devices.

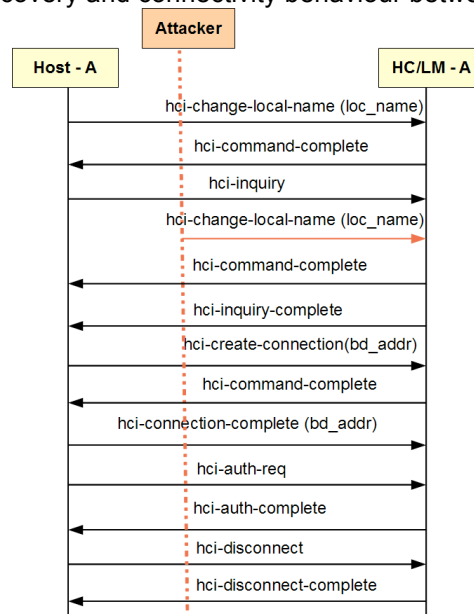


Figure 2: Sequence diagram – Session establishment and Bluestabbing attack

1) Bluetooth connection establishment property: Figure 2 shows the message sequences captured from the HCI layer of the master (car's Bluetooth device) while trying to achieve the Bluetooth connectivity with the slave device (mobile phone). Each Bluetooth device has a device local name, a user-friendly name to identify the different Bluetooth devices. This device name can be initially configured by each host by sending an *hci-change-local-name* message to the host controller (HC/LM-A). A Bluetooth device in discoverable mode can communicate or be visible to other Bluetooth devices if needed, it can be in non-discoverable mode. Devices which are in discoverable mode are only eligible to participate in the piconet. So when one Bluetooth device wants to connect to another one, it must go through certain steps to learn and authenticate

with the remote device. The master first finds the other devices which are in “discoverable” mode, and then performs an inquiry on each device by sending an *hci-inquiry* message. Thus the inquiry process gives the master a list of hardware addresses called *bd-addr* which are available to be connected and the important device feature information. The *bd-addr* is a unique address of a Bluetooth device, similar to MAC address of a network device. This address is needed for further communications with a device. Having received the slave addresses, the master can establish an actual connection with one or more of the devices it found via the paging process. During the paging process the master sends an *hci-create-connection* message to establish a connection with a particular slave (based on the parameter *bd-addr*). The connection is successfully established upon receiving an *hci-connection-complete* message. Authentication can be explicitly executed at any time after a connection has been established by an *hci-auth-req* message. And the established connection can be anytime detached by the master device by an *hci-disconnect* message [28].

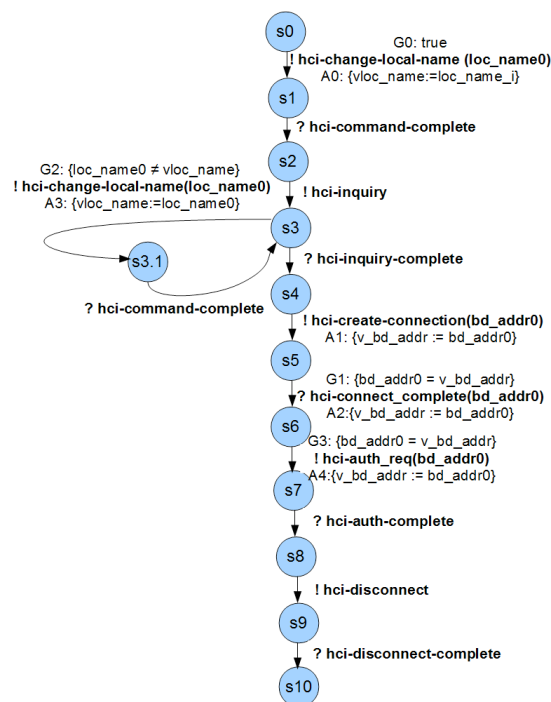


Figure 3: IOSTS model – Session establishment and Bluestabbing attack

2) Bluetooth attack - Bluestabbing: Usually, the list of discovered Bluetooth devices displays only the name of the located device, and it does not show the actual Bluetooth address. If the slave devices are familiar with the located device name they are in discoverable mode else invisible. The local name can be changed anytime by any one, hence the device is prone to Bluestabbing attack [29] as shown in Figure 2. In the Bluestabbing attack, the attacker impersonates as a legitimate user and modifies the Bluetooth device name of a legitimate user by resending an *hci-change-local-name* message with a badly formatted device name by causing the slave device to confuse during the device discovery phase (Inquiry). But this attack could be more severe, if the Bluetooth attacker modifies his own local device name as a legitimate user’s device name, and tries to establish a connection with the other Bluetooth device by capturing the passwords and sensitive information from the device.

For better understanding of the IOSTS formalism, we represent the Bluetooth behaviour along with the attack scenario in Figure 3. We observe that there is a transition from state S3 to state S3.1, a deviation from the regular scenario due to the *hci-change-local-name* message inserted by the attacker, during the device inquiry phase, resulting in the Bluestabbing attack. For explanation we have considered only few parameters *bd-addr*, *loc-name* corresponding to the Bluetooth protocol, but in practice there is no limitation in considering the number of parameters. In an IOSTS, all the values of the parameters and variables are represented

symbolically. For instance, the IOSTS tuple for the symbolic transition $t \in T$ from S_0 to S_1 can be expressed as below.

$$t : \langle S_0, (hci-change-local-name), true, ((vloc_name := loc_name_i)), S_1 \rangle$$

In the next section we introduce the symbolic execution of IOSTS based on the application to Bluetooth protocol.

2.2.1.2 SYMBOLIC EXECUTION

The symbolic execution (SE for short) of IOSTS serves two main objectives: (i) to use symbolic values for action messages and initialization values for IOSTS variables instead of concrete data values (ii) to obtain a tree-like structure which represents all the behaviours accepted by the IOSTS in a symbolic way. In Figure 4, the symbolic execution of an IOSTS is represented as a tree with different branches in which the vertices are symbolic extended states and edges are labelled by symbolic communication actions [30]. The branches depict the connection establishment and Bluestabbing attack in Bluetooth protocol as explained in previous section. The definitions related to symbolic execution of the IOSTS is provided in [31].

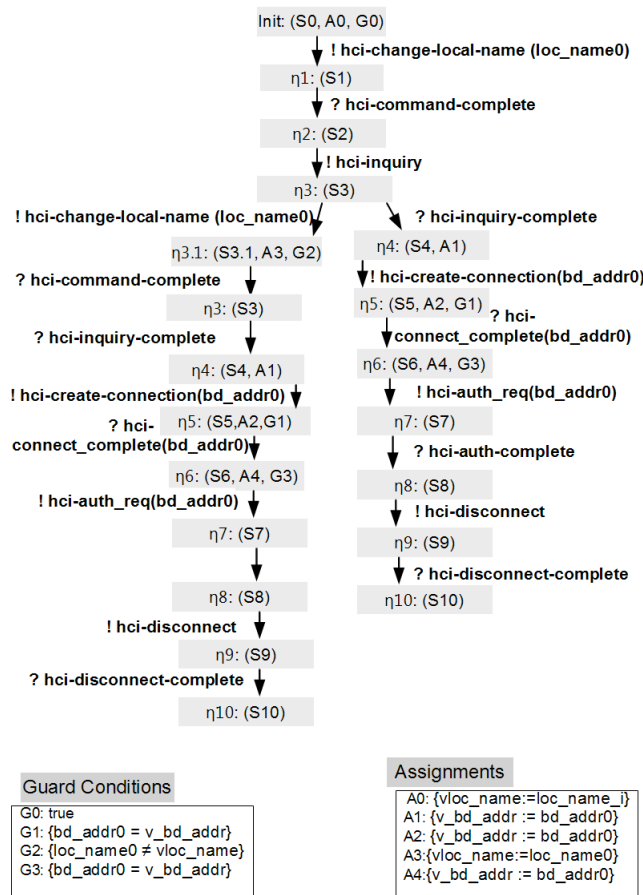


Figure 4: Symbolic execution of IOSTS

Now that we may obtain the traces of the symbolic execution of an IOSTS representing a protocol property with an eventual attack scenario, we define in the following a parametric trace slicing approach to passively test these symbolic traces on real network traces.

2.2.1.3 PARAMETRIC TRACE SLICING

Trace slicing is a widely used technique for analysing a real network trace [26]. The parametric trace slicing technique simplifies the real protocol execution trace into different slices based on the data portions of each event (i.e. packet) in the trace. The events corresponding to a particular valuation are grouped in the order they appear in the trace in a particular slice, and all the other events that are unrelated to the given valuation are dropped. The definition for *parametric traces* and then few terminologies related to our parametric trace slicing algorithm can be found in [31].

Definition 2. (Trace slicing) Given a parametric trace $\tau \in \Lambda^*$ and $v \in \Pi^{\mathcal{L}}$, a v -**trace slice** represented by $\tau \downarrow_v$, is the sequence of actions observed in τ , i.e. $\tau \downarrow_v \in \Sigma^*$, and defined as: $\tau' \langle a_i, v_i \rangle \downarrow_v = \tau' \downarrow_v a_i$ when $v_i \sqsubseteq v$ and $\tau' \downarrow_v$ when $v_i \not\sqsubseteq v$.

It means that for a finite parametric trace τ represented by $\tau' \langle a_i, v_i \rangle$, the trace slice for v can be obtained under two different cases. If v_i is less informative than v , then the action a_i is added at the end of the trace slice else we leave the trace slice undisturbed as defined above.

Example 1 Consider a sample Bluetooth trace, $\tau = !hci-inquiry?hci-inquiry-complete!hci-create-connection(bd-addr1)!hci-create-connection(bd-addr2)?hci-connect-complete(bd-addr1)$.

For simplicity, we represent only the parameter values when writing valuations, that is, instead of $\langle bd-addr - >bd-addr1 \rangle$ we denote by $\langle bd-addr1 \rangle$. Applying our trace slicing algorithm [31] on τ based on the parameter $bd-addr$ we obtain the Table 1

Table 1: Slice table for a sample Bluetooth trace τ .

Valuation (v)	ts – the v -trace slice
$\langle bd-addr1 \rangle$!hci-inquiry?hci-inquiry-complete!hci-create-connection ?hci-connect-complete
$\langle bd-addr2 \rangle$!hci-inquiry?hci-inquiry-complete!hci-create-connection

In this section, we defined how we can obtain the trace slices by applying our trace slicing algorithm. In the next section, we discuss how the obtained parametric trace slices are used for evaluating a Bluetooth property defined as an IOSTS.

2.2.1.4 TESTING THE IOSTS PROPERTY ON REAL EXECUTION TRACES

2.2.1.4.1 Evaluation of a Property on Trace slices

In our evaluation approach, we evaluate the control and the data portions of the messages observed in the symbolic traces. The evaluation is done for each slice against the symbolic traces based on the Table 2. The verdicts *Pass* and *Fail* are provided for the test of the conformance property while *Attack-Pass* and *Attack-Fail* are dedicated to the test of the security property. *Inconclusive* is emitted if we cannot firmly decide (e.g. in case of a too short execution trace).

Table 2: Evaluation table for each trace slice.

AttackSeq	Control Portion	Data Portion	Verdict
0	✓	✓	Pass
0	✓	×	Fail
1	✓	✓	Attack-Pass
1	✓	×	Attack-Fail
0 or 1	×	-	Inconclusive

An algorithm based on the above evaluation logic is presented in [31]. Finally, based on the verdicts obtained on the slices, we may define the final verdict of the property testing on the entire real trace by:

- **PASS**, if $(\forall Verdict(tsi) = Pass)$
- **Attack-PASS**, if $(\exists Verdict(tsi) = Attack-Pass)$
- **FAIL**, if $[(\exists Verdict(tsi) = Fail) \wedge (\exists Verdict(tsi) \neq Attack-Pass)]$
- **INCONCLUSIVE** otherwise.

2.2.2 Innovation with respect to the STOA

From our knowledge, there are currently no works tackling Passive testing/Monitoring based on IOSTS without any awareness on the states of the execution traces, moreover

- A novel passive testing approach based on parametric trace slicing and symbolic execution techniques was developed.
- The definition of an algorithm for parametric trace slicing by taking into account the data portions contained in the trace events.

The definition of a novel algorithm to check whether an IOSTS property is satisfied on a real execution trace. We also demonstrate that security attacks can be monitored by our approach.

2.2.3 Applications and contributions to case studies

Symbolic passive testing approach is illustrated by its application to a set of real execution traces extracted from a real automotive Bluetooth framework with functional and security properties. As part of our contributions to the **automotive case study** we were able to perform successfully a post-mortem/passive testing analysis in testing functional and vulnerability/ attack patterns.

2.2.4 Metrics and results

For the experiments, few traces were provided by the Dornier Consulting company for the automotive case study. Each of the Bluetooth trace obtained had different device local name. In order to evaluate the efficiency of our approach, we performed our experiments in two ways: with unmodified traces and by manually introducing errors and also by introducing few fake messages to create an attack scenario in the real trace. For example, in the traces 1,3 and 5 we tried to modify manually the parameter bd-addr in Bluetooth message like hci-connect-complete so that we can obtain Fail verdict. Introducing error in the message caused the guard conditions to fail. In order to detect the attack scenario explained in the Section 2.1.3, we introduced few fake messages to the traces 2,4,6 and 7 to create a Bluestabbing attack scenario. The attacks introduced were also correctly detected by our tool. The resultant outputs are shown in Table 3. The verdicts obtained before and after introducing the errors are provided in the Table 3.

Table 3: Prototype tool results on sample Bluetooth traces

Trace	No. Messages	No. Slices	Traceoutput with- out errors				Trace Outputs with er- rors and attacks				
			P	F	I	Final	P	F	I	AP	Fi- nal
1	81	2	1	-	1		-	1	1	-	F
2	89	3	1	-	2		-	-	2	1	AP
3	81	2	1	-	1		-	1	1	-	F
4	81	2	1	-	1		-	-	1	1	AP
5	81	2	1	-	1		-	1	1	-	F
6	81	2	1	-	1		-	-	1	1	AP

7	81	2	1	-	1		-	-	1	1	AP
---	----	---	---	---	---	--	---	---	---	---	----

2.2.5 Tools support

A prototype tool model, **TestSym-P** has been developed by IT during the project period of DIAMONDS to apply our approach of Symbolic passive testing. The proposed algorithms for trace slicing and evaluation logic as described in Section 2 are implemented in this prototype model. A detailed description of the tool is provided in D5.WP3. Our prototype and the sample files used for the experiments can be found at <http://www-public.it-sudparis.eu/~mouttapp/TestSym.html>.

2.2.6 Exploitation and dissemination

The results of this work have been published and presented in several international conferences and summer schools as mentioned in the dissemination activities provided in D5.WP5 deliverable.

In addition, IT (Institut Telecom) will exploit the results of DIAMONDS internally, in order to advance relevant research in IT and the enrichment of the courses. On the other hand, external exploitation of the results will include efforts for the dissemination of the results in other fields (such as protocol engineering and telecommunications) and amongst other academic partners and companies, as well as publication of the results in prestigious conferences, journals (as mentioned in the dissemination report) and talks delivered in summer schools and workshops.

2.3 DISTRIBUTED AND COOPERATIVE INTRUSION DETECTION

2.3.1 Description

A distributed and cooperative intrusion detection approach is proposed for Wireless Ad-hoc Networks [34]. The proposed intrusion detection architecture is illustrated in Figure 5.

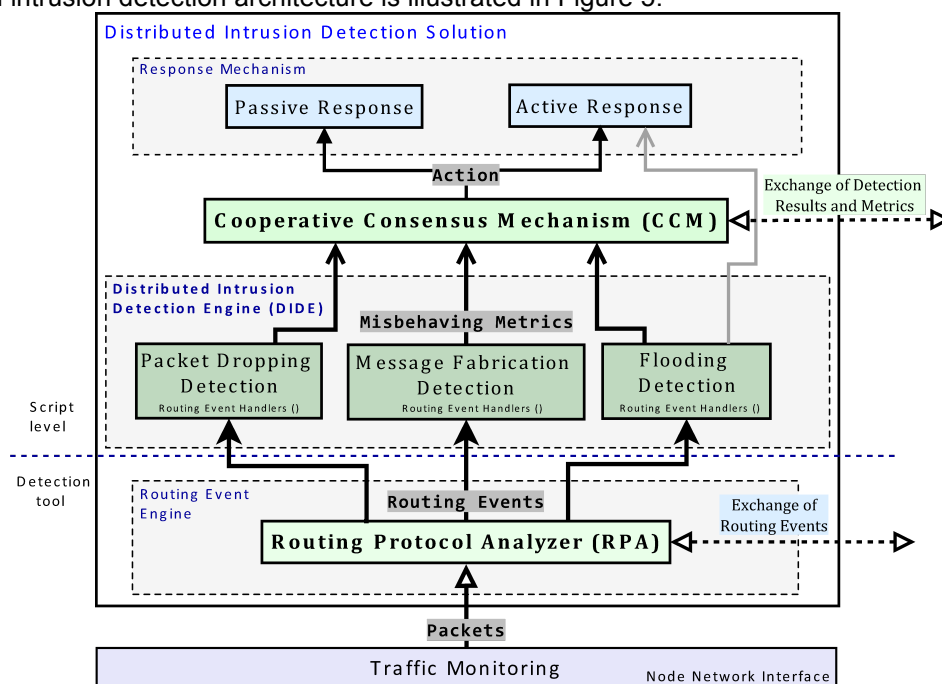



Figure 5: Distributed intrusion detection architecture

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 21 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public


In the approach, an Intrusion Detection System (IDS) tool [35] runs individually on each node to passively monitor the network traffic and eventually detect routing attacks in collaboration with the neighbouring nodes. Specific traffic filters are implemented for the IDS tool in order to select the appropriate routing packets being transmitted on the node network interface. Then, each node IDS tool concurrently analyses the collected routing traffic and generates routing protocol events, which are processed by the own node and subsequently exchanged with direct neighbours to minimize the message overhead when monitoring the neighbourhood behaviour. For that purpose, a Routing Protocol Analyzer (RPA) module is developed, which examines each routing packet and produces respective Routing Events according to the protocol behaviour. In the work, BATMAN routing protocol is employed as case study, and therefore BATMAN [36] specification is used to define the Routing Events. However, the RPA module is quite customizable and flexible, and can easily accommodate other routing protocols as well.

The core part of the distributed intrusion detection architecture is the Distributed Intrusion Detection Engine (DIDE) component. This module processes the Routing Events produced by the RPA module and also the Routing Events received from neighbouring nodes to cooperatively detect routing misbehaviour intrusions. Routing Constraints are defined and extracted from the routing protocol behaviour and are integrated into DIDE component. The BATMAN specification is used to formulate the Routing Constraints, nevertheless additional routing protocols are similarly supported. Part of the routing attack detection algorithms are implemented in the DIDE. Based upon the Routing Constraints, the DIDE component periodically calculates Misbehaving Metrics. The Misbehaving Metrics indicate if possible malicious routing behaviour is occurring on the node network interface or in the neighbourhood, i.e., if the node or a neighbour is originating malicious routing traffic, for instance, the malicious node is transmitting fake routing packets to its neighbours. The collaborative intrusion detection procedure, which is carried out by the DIDE component, is executed in real-time by the IDS tool having low computational overhead and reasonable message overhead, and it is quite efficient with regard to hardware resources consumption.

A Cooperative Consensus Mechanism (CCM) module is triggered by the node IDS if any evidence of malicious routing information is found by the IDS agent. The CCM module inspects the Misbehaving Metrics provided by the DIDE component, such as rate of inconsistent routing packets received by the node, and confirms if abnormal routing behaviour detected by DIDE is a routing attack or not based on thresholds. Then, the node IDS warns the neighbouring nodes about the detected routing intrusion. Before making the final decision, the node IDS consults its neighbours. Thus, the CCM module shares Intrusion Detection Results with neighbouring CCM modules to confirm that its neighbours diagnosed the same routing attack to finally make a collective decision about the suspicious node. If the majority of nodes detected the intrusion, a consensus is reached among the neighbouring nodes on the suspected node. The proposed Consensus Algorithm coordinated among neighbouring nodes is able to efficiently detect malicious routing traffic in a neighbourhood and the source of the intrusion with good precision in real-time.

A threshold-based scheme that is used along with CCM module to enhance the accuracy of analysis of Misbehaving Metrics. The threshold mechanism takes into consideration the rate of packet loss between communication links of nodes to differentiate attack detection rate from false alarm rate in order to achieve a better true positive rate and low false positives. The threshold-based mechanism results in smaller false positives and also provides better efficiency in terms of intrusion detection rate.

The efficacy of the distributed intrusion detection approach is demonstrated in a virtualized Ad-hoc network environment that consists of VMs interconnected. The VMs connected with each other are equivalent to radio nodes executing the distributed intrusion detection solution in an Ad-hoc topology. In this platform, the adversary nodes are represented by using VMs which execute attack injector tools. These attacking tools are employed to implement the most common routing misbehaviour attacking techniques such as routing message fabrication and packet dropping attack variants [37]. In that manner, the performance of the distributed intrusion detection algorithms is satisfactorily evaluated in the virtualized Ad-hoc network platform. The experimental results show that the distributed intrusion detection method achieves high detection rate for different types of message fabrication and packet dropping attacks with no false negatives and low false positive rate. The IDS implementation presents good effectiveness and high level of reliability in the role of detecting different routing attacks, and yet incurs small computation overhead, i.e., CPU and memory over-

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 22 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

head, and acceptable communication overhead for the radio devices. Then, the intrusion detection solution achieves scalability having good performance in terms resources consumption with practical assumptions.

2.3.2 Innovation with respect to the STOA

Our IDS approach is mainly based on specification-based intrusion detection technique since we specify Routing Constraints, which define the correct protocol behaviour, so we can detect malicious behaviour that violate these constraints. The reason for using Routing Constraints is they are optimized to cover the most common routing attacking methods of Wireless Ad-hoc Networks: message fabrication, packet dropping, and message flooding.

However, our IDS approach innovates on the use of routing events, that are defined based upon the protocol specification and are generated as a result of the analysis of each routing message. By making use of routing events together with Routing Constraints, we can identify most of attacks of Wireless Ad-hoc Networks in a reliable and collaborative way since routing events are analysed in parallel by the node and its neighbouring nodes to compute misbehaving metrics.

Furthermore, we innovate by integrating a threshold-based mechanism to distinguish between accidental packet loss of links and malicious packet dropping. Hence, our intrusion detection approach provides better detection accuracy with no false negatives while limiting the number of false positives.

Moreover, the approach is able to identify any new attack that makes use of these common attacking methods without the need of updating the intrusion detection engine.

2.3.3 Applications and contributions to case studies

The main contribution of this work is the implementation of a complete distributed and cooperative intrusion detection solution for an Ad-hoc Radio Network. The IDS approach can be deployed at the Radio Link Control (RLC) layer of Thales Radio Protocol, which is used to route data within the network with single relay station. In addition, the IDS solution is applicable to the Radio Service Network (RSN) layer since this layer performs route search in the internal addressing plan on RLC request supervising the local routing and executes the routing protocol itself.


The IDS software focuses on efficiently detecting insider intrusions in real-time by collaboratively monitoring the network activity of the nodes and by exchanging suspected traffic information. To that end, we propose a Routing Protocol Analyzer (RPA) that examines each routing packet and produces respective Routing Events according to the protocol behaviour. The RPA module is quite customizable and flexible, and can accommodate other protocols.

In addition, routing attacks against the nodes of a Wireless Ad-hoc Network are extensively studied in order to reveal security breaches of the protocol. We provide analysis of different routing attacks and demonstrate the feasibility of the attacks in detail. We show how a unique compromised node is capable of manipulating the routes of the rest of the nodes and redirect the traffic to the compromised node. In order to validate the analysis, we implement these attacks in an emulated Ad-hoc network environment composed of virtual machines (VMs), which represent the nodes executing the routing protocol. We perform extensive experiments on emulating routing attacks to check the protocol security mechanisms.

We also present efficient mechanisms to detect routing attacks and identify the source of attack, i.e., the malicious node, which makes use of the traces produced by each node in the network.

2.3.4 Metrics and results

We obtained an effective detection rate, i.e. true positive rate, of 100% for nodes relying exclusively on local Routing Events, i.e., they are not affected by packet loss, and a detection rate of 96% for nodes affected by packet loss rate. The false positive rate is 1% for nodes undergoing packet loss rate. The false negative rate

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 23 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

is 0% since all malicious packets are precisely detected by the nodes. Therefore, the distributed intrusion detection mechanism does not yield false negatives.

2.3.5 Tools support

The complete distributed intrusion detection solution is implemented in Bro IDS v.1.5.3 [5], then allowing Bro tool to detect routing intrusions in Ad-hoc networks.

2.3.6 Exploitation and dissemination

The obtained results of this work have been published and presented in several international conferences and summer schools as mentioned in the dissemination activities provided in D5.WP5 deliverable.

In addition, IT will exploit the results of DIAMONDS internally in order to advance relevant research and the enrichment of the courses. On the other hand, external exploitation of the results will include efforts for the dissemination of the results in other fields (such as protocol engineering and telecommunications) and amongst other academic partners and companies, as well as publication of the results in prestigious conferences, journals (as mentioned in the dissemination report) and talks delivered in Summer schools and Workshops.


2.4 ANOMALY DETECTION WITH MACHINE LEARNING APPROACH IN ICS NETWORK MONITORING

2.4.1 Description

Intrusion detection refers to a technique developed to detect malicious network activity in a single host or in an entire network. Currently, Network Intrusion Detection Systems (NIDS) are not widely used in industrial control system (ICS) networks. This leaves a hole in the network defenses' of industrial sites. Network Intrusion Detection attempts to discover anomalous or malicious activity by analysing traffic in the target network. Intrusion detection systems can be divided to two subcategories: anomaly based, and signature based. Our technique focuses on anomaly based detection technique that leverages the determinism of ICS network traffic. Having a NIDS that is able to adapt to the particular environment of an industrial site would reduce the risk of production disruptions caused by anomalous incidents on the network. Machine learning approach offers important complement to traditional signature based detection techniques especially in relatively closed environments. ICS networks are restricted networks, not really closed, even if we would want them to be. The firewalls are typically full of holes created for system vendors to grant them maintenance access for the devices they have supplied. The connections for vendors and other personnel requiring exceptional access to systems residing on the control typically do not follow default policies in place for connections. However, the restricted nature of the ICS network does give the NIDS developer some advantages in addition to some specifics that need to be taken into account when designing the system.

2.4.1.1 Machine Learning

Machine learning performs well for network intrusion detection applications in a closed environment, but exposing it to the open world with varying amount of random traffic, or noise, adversely affects its usability. The factory networks for ICS that are functioning properly and without serious design flaws can be defined as nearly closed environments. When the factory network would be in a normal status without serious incidents there is typically very little noise. Again, if the network architecture is well defined and implemented, most of the network traffic on the ICS level of the network should be more deterministic than that of open networks such as the office networks. Events that cause noise include maintenance of the network, new devices being added to the network, devices that send various configuration and control messages, or malfunctioning devices in the network. Depending on the frequency of these types of events, it might be reasonable to require a report of these actions to the personnel responsible for maintaining the network monitoring systems. The network administrators could then either temporarily switch off the system, manipulate its parameters or set it to a learning mode.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 24 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

2.4.2 Innovation with respect to the STOA

Our innovation is to use IDS anomaly detection with machine learning in ICS environments. Traditional intrusion detection systems still rely mostly on signature-based techniques. Using anomaly detection for network traffic is not a new topic, but it has not been studied nor applied in the ICS context extensively. Also, there are no commercial anomaly detecting IDS tools available for ICS environments.

We developed an early prototype of a system leveraging machine learning for intrusion detection in ICS network context. Currently we are experimenting using Bro Network Security Monitoring, Rapidminer and Simple Event Correlator with ICS network trace files. The prototype of our machine learning tool is written with python.

2.4.3 Applications and contributions to case studies

Traffic captures from the industrial case study partners have been used as a training material for our machine learning system.

2.4.4 Metrics and results

The diversity of the Internet traffic makes anomaly detection with machine learning approach very challenging in traditional office networks. When the traffic flows and hosts are more regular, this approach is more feasible. Preliminary results show promising results; the learning system works very well because of the low number of false positives. However, our technique is by no means a plug-and-play system. The learning and configuring are challenging and careful tasks that have to be done individually for each environment.

2.4.5 Exploitation and dissemination

The technique and approach of VTT have been disseminated in international academic security conferences, journals and other suitable forums provided in D5.WP5 deliverable. The technique and traffic captures have been exploited our internal research projects. This approach has also provided expertise to maintain competence in the areas of network security and industrial control systems.

2.5 PASSIVE TESTING AND NETWORK MONITORING

2.5.1 Description

2.5.1.1 Instrumentation

When doing software testing, one half of the process is generating test cases and inputting them into a SUT (System under Test). The other half is determining how the SUT reacts on these inputs [24]. This is called monitoring or instrumentation [22]. The type of instrumentation depends on the SUT. Instrumentation for a Windows program is totally different than for a WLAN (Wireless Local Area Network) router [25]. On computer, it is relatively easy to install a debugger or another out-of-band instrumentation method, but on a WLAN router it is easiest to use in-band instrumentation methods [23].


Instrumentation techniques used in this project:

2.5.1.1.1 SNMP Instrumentation

SNMP stands for Simple Network Management Protocol. One purpose of SNMP is to monitor network elements [32]. So with the help of SNMP, it is possible to get some information about the system where the SUT is running. It is required that the system supports SNMP and that SNMP requires agents that support showing system resources. With SNMP, it is possible to see the current CPU load or currently available free memory [33].

2.5.1.1.2 Valid Case Instrumentation

A step forward from the UDP connection and the TCP connection instrumentation is the valid case instrumentation. Here the idea is that after each test case, one valid case, which has only valid input, is sent. If an

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 25 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

answer is received for this valid case, the SUT is still operating, and if not, then the last test case has caused a crash

2.5.1.1.3 External Instrumentation

When you are testing, you can use instrumentation to define external commands and scripts to be executed. Such external scripts and commands are frequently used to make it easier to perform functions, such as checking SUT health, performing specific initializations, automating test result reporting routines and performing SUT and environment recovery. Commands and scripts can be called before and after each test case or test run. They can also be used to instrument, that is to check, SUT health. In case the instrumentation fails, a script can be called to recover the SUT.

2.5.1.2 Analysis

2.5.1.2.1 Network Analyzer

The Codenomicon Network Analyzer records test behaviour and automatically visualizes network traffic. It provides testers with a clear picture of actual network traffic. The solution can store and process terabytes of real-time network data, and automatically visualize all network flows and identities, making it extremely easy to search and pinpoint troubling network issues in any IP –based network. The Codenomicon Network Analyzers reveals what is really happening in the network helping testers decide what should be tested and how. The Analyzer’s collaborative framework enables users to model actual network traffic, to troubleshoot networks and to debug any type of network communication from multiple locations. The problematic messages and sequences can be easily extracted to tools like Codenomicon Defensics for reproduction and testing, or to open source tools like Wireshark for closer inspection. The effective collaboration tools and the synthesis of network documentation and network traffic enable users to analyse networks faster and more thoroughly than with any other tool. The measured peak processing has been more than 70.000 pps, with no packet loss.

2.5.2 Innovation with respect to the STOA


There was no major innovation on this area, but lot of learning and need for better approach was identified. Implementation is starting but not in the scope of the Diamonds project.

Instrumentation needs to be a separately evolving, loosely coupled mechanism of different instrumentation agents and a centralised reporting component. When fuzzing testing is applied in wider technology areas and different industries, the main impact is in different protocols and in instrumentation. The instrumentation cases range from digital output measurement (industrial automation) to process debuggers and stack traces (high end networking elements like routers and general computer software). In many cases our flexible external instrumentation can be used to accept the verdicts from different sources, but the performance penalty caused by the implementation is often unacceptable. Clearly a better architecture needs to be defined and implemented.

Challenge for the centralised reporting is to prioritise conflicting instrumentation sources. For example in industrial automation the controlled output is highest priority, and any non-conformance on the output signals a failure even if the valid case instrumentation passes. The simple solution is to signal any source failure as a failure for the test case regardless of the other inputs. Synchronising the different instrumentation sources can be challenging, if the instrumentation source is on the target machine (system under test), e.g. debugger, the output synchronisation to a single test case may be unacceptably slow. Time based synchronisation requires clocks to be synchronised, and ability to combine the different logs in the reporting component based on the timestamps. The value of additional information for a verdict increases. The mechanisms must be able to store large amounts of debugger data and the actual test case information.

Also it was identified that pass/failure is a policy dependent verdict. There are cases when warnings should be given, either due to load in the target that is high but not critical. A new area of testing may be in question when verdicts are given for detected un-secure algorithms or if sequence execution continues when authentication values were anomalised. The expansion of the verdicts impact reporting and may need user control what kind of verdicts are wanted to be reported.

Utilisation of network analyzer is needed for system testing. The main concern in systems is that an input causes a much larger output and it can cause resource overload in some other parts of the system. Also a

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 26 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

system can degrade to state where it is less capable of handling the load, and causes still extra load when it attempts to control the load and divide the processing to other systems. Analyzer can be used to identify the bottlenecks on system level, in other parts than the direct path from tester to device-under-test. Analyzer usage still complicates the results handling, there needs to be a mechanism to identify loads in different parts of a system that is related to the testing. This area requires further study, expert based manual work is needed at the moment.

2.5.3 Applications and contributions to case studies

All the methods described in chapters “Instrumentation” and “Network analysis” were applied in Telecommunication case study and Industrial Automation case study by Codenomicon (please refer to D5.WP1).

2.5.4 Metrics and results

The expanded definition and usage of instrumentation above impact metrics and results. In case studies the results are still pass/fail based.


See Telecommunication and Industrial Automation Case study reports.

2.5.5 Tools support

The above learnings are only entering design phase, and are not yet available in tools. Implementation will aim to allow the usage of the learnings in tools regardless of the suite version once an implementation is finalized.

2.5.6 Exploitation and dissemination

All exploitation results are under Fast Exploitation provided in D5.WP5 deliverable.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 27 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

3. ACTIVE SECURITY TESTING TECHNIQUES

3.1 ATTACK PATTERN BASED TESTING

Making faults twice is definitely something that should not occur. This holds especially for programs in case of security. A successful well-known attack should not be successful anymore. Unfortunately, this does not hold in practice. One reason is that testing for vulnerabilities is hardly be tightly integrated within the software development process. In order to overcome this shortcoming we suggest a method that is based on a model of attack patterns, which can easily be automated and thus tightly integrated even within today's agile processes used for developing programs.

3.1.1 Description

Attack pattern based testing is a model-based security testing technique, which uses patterns of known attacks on web applications. Its goal is to detect vulnerability of a system under test by traversing and executing an attack model against a SUT. Although the source code of the application is known (white-box), the goal of this approach is not to test its structure but only to give information about potential security leaks by trying to breach the software.

The main topic here is to recreate the circumstances in which a malicious action occurs by means of attack patterns. These describe the goal, pre- and post-conditions and actions to be executed in order for the attack to be realized [1].

Several categories of attacks against web applications exist, the most common and malicious being enumerated in [2]. The model covers the pattern of one type of attack so the SUT is tested against one type of attack per pattern. SQL injection (SQLI) and Cross-site scripting (XSS) are still the most common attacks on web applications so the given approach initially takes into account these two attack types. The defined modelling language relies on the UML notation [3] but is in fact an adapted state machine.

Once a pattern is fully specified and modelled, it will be executed automatically with only small user interaction because some program specific attributes have to be defined by the tester. The interaction also gives the tester the possibility to guide the execution through alternate model paths and he may also stop the execution and may manually choose individual parameters by himself. Also, an important fact is that the model itself is expandable so the tester may have the possibility to add additional states, transitions and methods afterwards.

3.1.2 Innovation with respect to the STOA

The main innovation of our approach is the inclusion of attack patterns in model-based testing. Other methods include the use of a model of the correct behaviour of the SUT so test cases are created in runtime in order to check the consistency of the real applications with respect to the model. These methods rely on the concept to detect faulty behaviour of the application and can either be classified as white-box or black-box testing approaches.

Attack patterns are modelled and executed against a program. Test case generation is also included within the overall concept. The idea of the entire approach is to carry out the role of the attacker by trying to imitate malicious behaviour in form of the attack model. Actually, the attack pattern execution as well as test case generation is fully guided by the implemented attack model.

The other innovation is due to possibility to integrate the approach within today's agile software development processes where full automation of testing is required. This is possible because the test cases are generated automatically when traversing the attack model given in form of an UML state machine. Hence, the attack-pattern based approach allows for automated security testing.

3.1.3 Applications and contributions to case studies

In the context of the DIAMONDS our attack-pattern based approach contributes to all case studies that include web interfaces. We have not tried to apply the approach directly to the case studies due to the late beginning of the Austrian DIAMONDS project but have done a first empirical evaluation, which we describe later in this chapter.

As mentioned above, this method makes use of UML state machines [4] and uses its elements, states and transitions in order to realize all needed information for the purpose. Figure 6 shows a model which implements an SQL injection attack pattern and executes it against a custom system under test.

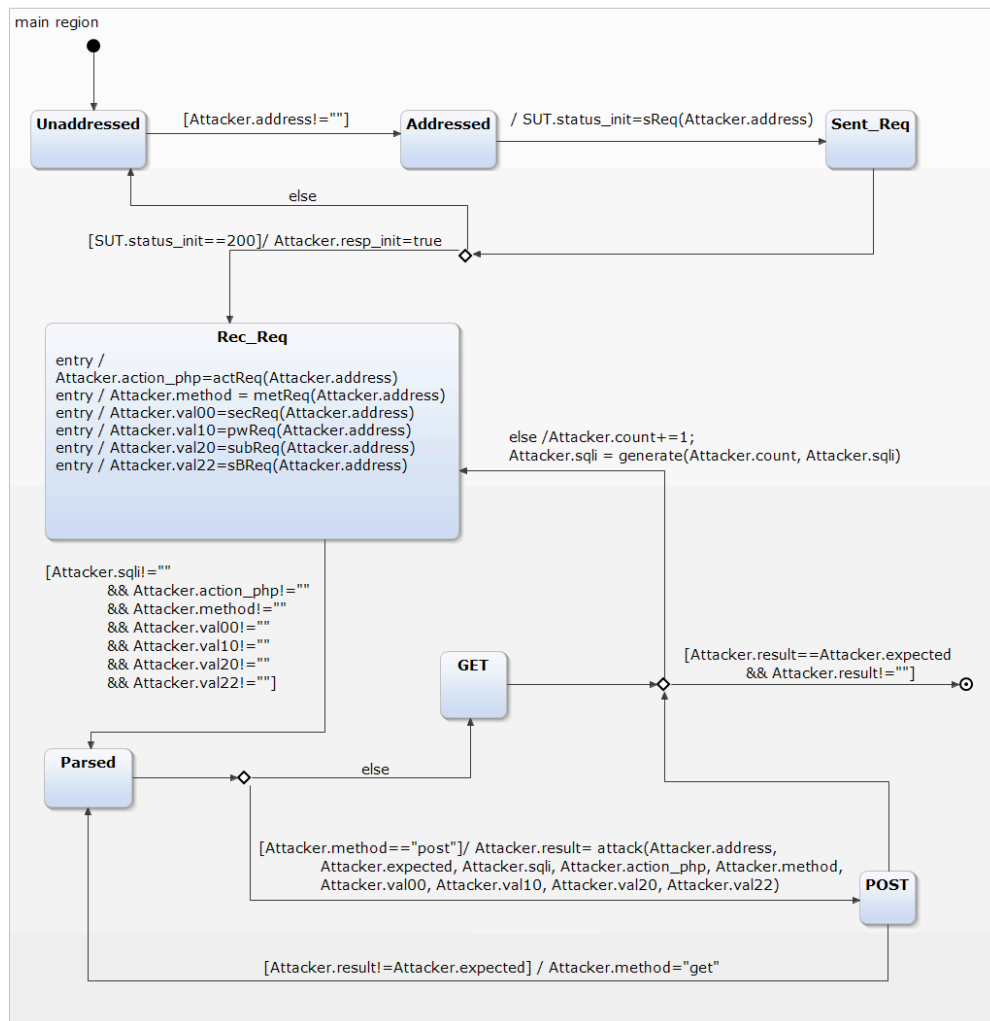


Figure 6: SQLI Attack model

A state machine Σ is defined by the following formula:

$$\Sigma = (S, V, M, G, A, T, s_0, F)$$

where

- S represents a finite, non-empty set of states.
- V is a set of variables.
- M defines the set of methods, which manipulate variables.
- G is the set of guards. They represent pre conditions in transitions.
- A is the set of actions. Actions are post conditions in transitions and manipulate values of variables.
- T represents the transitions between states. They may have pre- and post-conditions.
- s_0 is the initial state of the model.
- F is the final state. The execution of the model ends there.

Every state in the model may have specified some entry, exit or inside-processing action. The entry actions define actions to take place when entering the state from outside. Furthermore, functions can be called inside the state and exit actions specify activities to be done before deactivating the current state, thus entering the next.

Figure 7 depicts a typical transition label between states. Each one of them may have a guard and an action although this is not obligatory.

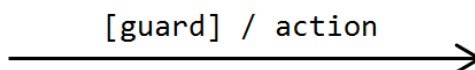


Figure 7: Transition between states

The guard acts as a pre-condition and specifies what has to be satisfied in order for the action to take place. The action is the post condition which changes some value and activates the successor state.

Moore et al. [1] define attack patterns, which include the common elements: goal, pre- and post-conditions and actions. This information is needed for the tester to implement the attack model for a specific malicious action like SQLI or XSS.

The goal defines the general motivation for the pattern, in this case to detect security vulnerability in a SUT so improving the injection script may access eventually sensitive data. The pre-condition specifies necessary actions for the attack to take place, usually the access to the application and the possibility to submit input data. Post conditions specify the realization of the goal and are usually the successful detection or exploitation of vulnerability. Actions enumerate all mandatory steps in order to get from the initial state made possible by the pre-condition to the final state that results in the post condition and goal.

Figure 8 depicts the GUI of a simple PHP application to be tested against SQL injection. It consists of a login field with two input fields and the submit button. The SUT is deployed on an Apache Server and has a MySQL database with several user credentials. The user may authenticate him by submitting a correct username and password. The goal is to access data stored behind the application in the database, thus detecting possible SQLI vulnerability.



Figure 8: GUI of the SUT


The model in Figure 6 depicts all necessary actions in order to satisfy the goal. The entire approach including all attack pattern model elements and data is implemented in the open-source toolkit YAKINDU Statechart Tools⁶. It offers a modelling environment for overall systems, which are based on state machines and may be installed as a plug-in in Eclipse.

The variable and method declarations are specified in YAKINDU and all methods are implemented in Java and may be called from the model either as parts of states or transitions who in turn may process data and return values for further execution. Guards, actions and variables are specified as expressions or functions inside the states or as parts of transitions in the model.

In order for the attack model to communicate with the application, HTTPClient⁷ is implemented in Java. The interaction occurs by means of sending and receiving of HTTP messages between the subjects. HTTP requests are crafted in the client with specific values and then sent to the URL address of the SUT. When receiving the application, the server sends a response to the client who parses the message and extracts its

⁶ <http://statecharts.org/>

⁷ <http://hc.apache.org/httpcomponents-client-ga/>

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 30 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

data from header and body of the response. The extracted data is needed in order to proceed further in through the model because they contain variable names, e.g. names of input fields, and are used as parts of other states or transitions.

Before execution, some data is mandatory to be specified by the tester. For example, the initial input string for SQLI (x' or $'x' = 'x$), URL address of the application (usually the *localhost* address) or the expected output of the test case.

The other important topic is the test case generation. The initial execution of the attack model is in fact the first test case. If the input string fails to detect SQLI vulnerability, the final state is not entered but another transition is taken, which acts as a test case generator in the model. For every test case, there is a predefined variable *count*, which is initially set to zero, which is incremented with every unsuccessful test case. So if the initial string does not succeed, one is added to *counter* and it is submitted with the input to the *generate* method. In this method, another sub-method is called according to the value of *count*, and these sub-methods return another input string. When using the SQLI pattern, the next test case may be to map all white spaces inside the input string by adding comments between symbols (thus $x'/**/or/**/'x'/**/=/**/'x$). This new string will return to a previous state in the model and the last attack steps will be traversed again. So this new input acts in fact as a new test case. If this new string also fails, then the counter is increased again and another sub-method is called inside the generator function. The next test case could be to URL-encode the string because the SUT may escape all apostrophes from the incoming SQL query so the new input becomes $x\%27$ or $\%27x\%27 = \%27x$ if URL encoding the initial string. In fact, the tester adds test case generation methods himself by taking into account SQLI related methods. The strategy can be also chosen freely (e.g. grammar-based, mutations, additional encodings etc.).

But if the client encounters a string in the HTTP response after submitting the injection which equals to the expected output string, then the test case is successful and model execution terminates in the final state. The generation and execution continues as long as new input strings are generated.

We now the execution of the attack model from Figure 6 is explained. The execution of the attack pattern from the state machine begins with the initial state. The guard from the transition starting at the *Unaddressed* state, activates the second state. Now, a HTTP message is crafted by taking the given URL. The state *Addressed* remains active until no address is specified.

Afterwards the execution proceeds to the *Sent_Req* state. This state is left when a boolean variable for signaling that a response has been accepted and having the status code 200 is changed. If that is not the case, the activity returns to the first non-initial state, so the tester may enter another URL. Otherwise, the next state becomes *Rec_Req*. This state is very important because it holds several variables and methods, which are used to create a valid HTTP attack message by getting all preconditions satisfied, i.e. the valid names of input fields so it can be guaranteed that the attack information is sent to the right place. Because all specified variables do not have any value, several methods are called in order to extract them from the response. Besides the input fields, the concerned information is about the HTTP method, URI, used technology etc.

After all variables have been assigned, a value *Parsed* becomes active. Then the malicious request is crafted using the extracted data from the previous step. This request is submitted to the application according the HTTP method. If a new response arrives and one of its body information matches the expected string, the execution reaches the final state. In this case we have successfully accomplished the attack. Otherwise, the counter is incremented and another SQLI input is generated within the state *Rec_Req*, which becomes active again. Now the checking and attacking process will be repeated over and over again until the attack is successful or no further SQLI strings can be generated. It is important to mention that only a successful attack accordingly to the user specification can be automatically detected. However, it might be the case that the attack invokes another unexpected behaviour on side of the SUT, which might not be treated as failure.

The execution of the test model reveals the faulty behaviour thus it is shown that the SUT is vulnerable against SQLI attacks. This case study demonstrates that it is possible to model attack patterns using an adapted UML state machine model. It is also worth mentioning that the model might comprise several different test cases, which are carried out during traversing the states. Testing can be automated using the model

making it applicable for a test driven development process. By modelling attack instead of the behaviour of the model it can be also ensured that a SUT has at least limited capabilities of preventing successful attacks that are well known.

This approach can be used in order to define an attack pattern model for other attacks.

3.1.4 Metrics and results

We (TU Graz) tested our approach using three different web applications, which are meant to be tested by initiates as well as security professionals, namely the *Damn Vulnerable Web App (DVWA)*⁸, *Mutillidae*⁹ and *Bodgeit*¹⁰. The first and second program contains multiple security levels and offers more exploitation possibilities. Bodgeit on the other hand has no additional levels and focuses more on SQLi and XSS. For the empirical study we apply our approach against all available security layers in order to exploit SQLi as well as both reflected and stored XSS vulnerabilities. For this purpose we made use of a model, which is as much abstract as possible and only slightly differs for every tested application. So it is assumed that every application can be modelled almost the same for the every attack pattern.

Figure 9 depicts the execution model for DVWA.

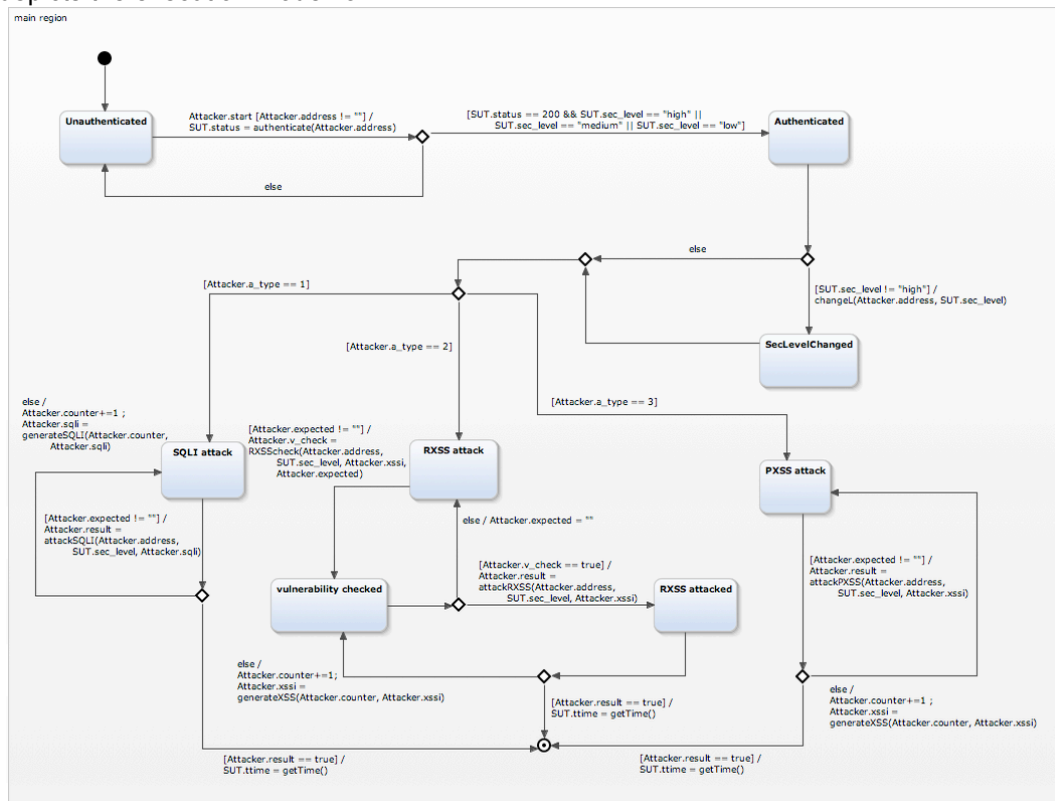


Figure 9: Attack pattern model for DVWA

In order to reduce the overall execution time, the number of states is kept smaller than the ones from Figure 6. Also, the model contains all three attack paths (one for SQLi and two for XSS) but only one type is actively executed during run-time, according to the choice of the tester.

The execution timer for the model begins with the calling of the first method (*authenticate()*) and ends in the final state. If no vulnerability is triggered, the execution remains stuck in the model so the time is not measured further. In fact, we take into account only those test cases, which lead to a positive result, i.e. a breach of the security. We used a collection of 33 SQLi input strings, which differ slightly w.r.t. several security lev-

⁸ <http://www.dvwa.co.uk/>

⁹ <http://www.irongeek.com/i.php?page=mutillidae/mutillidae-deliberately-vulnerable-php-owasp-top-10>

¹⁰ <https://code.google.com/p/bodgeit/>

els. There are most common inputs as well as more advanced examples that use special symbols, comments, escape special symbols, key words, UNION statements in order to guess the number of fields and the combination of these methods. On the other side, we used 17 XSS input strings for both types of this attack and also symbol escapes, URL encodings, keyword escapes etc. but also combined all of the input types.

For our empirical evaluation the attack strings were executed randomly. We summarize the obtained results in Table 4, which also includes the average values of the captured time for all test runs and applications under test.

Table 4: Execution time for DVWA, Mutillidae and Bodgelt.

Application	Type of attack	Security level	Average execution time (s)
DVWA	SQLI	low	0,97
		medium	2,28
		high	<i>not found</i>
	RXSS	low	1,41
		medium	2,39
		high	<i>not found</i>
	SXSS	low	0,92
		medium	0,90
		high	<i>not found</i>
Mutillidae	SQLI	low	1,39
		medium	4,16
		high	<i>not found</i>
	RXSS	low	1,66
		medium	2,31
		high	<i>not found</i>
	SXSS	low	1,94
		medium	2,74
		high	<i>not found</i>
Bodgelt	SQLI		4,49
	RXSS		0,99
	SXSS		0,83

The difference in the results may have resulted because of the fact that every program implements other filtering mechanisms. Unfortunately, it was impossible to detect vulnerability at the highest security level. Hence, the test case generation strategy has to be adapted for this purpose, which includes the use of evasion techniques for more advanced filtering mechanisms like the use of HTML special characters etc.


It should also be mentioned that the execution time of the model is far slower than the usual execution of Java code. For example, if we add another state in the model (e.g. into the SQLI path), the average time of medium SQLI exploitation lasts 2.49 s. By adding four states, we get 2.55 s. If we implement additional ten states, we obtain an average time of 4.42 s. On the other side, when adding choices and additional method calls into the model, there is almost no effect on the execution time. Hence, we conclude that the number of states has the highest influence on the overall execution time.

3.1.5 Tool support

The model of the attack pattern is formalized using the UML modelling language and the entire system is integrated into the Eclipse development environment. All methods are implemented in Java by relying on additional libraries like jsoup¹¹ for parsing the HTTP responses and search for specific elements like *scripts* or *inputs*. Also, the framework WebScarab¹² is used for manual interpretation of the communication between the client (attacker) and the server and the observed overview may help with the implementation of software specific elements like names of input fields etc.

¹¹ <http://jsoup.org/>

¹² https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 33 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

All methods are implemented in Java and are called by traversing an attack path, which acts as a test case. These methods in turn calculate new values, which are used further in the execution process. If previously active states are activated again, the methods may be used again to process new values, either intermediate ones or the actual testing input string. If a vulnerability is exploited, the execution terminates in the final state, thus concluding the attack pattern based testing approach.

3.1.6 Exploitation and dissemination

In [5] the authors give an overview of some state-of-the-art model-based testing approaches. They give an overview about several techniques by relying on black- as well as white-box testing methodologies and describe techniques like fuzzing. The authors also propose a model-based mutation testing approach by using a model of a specific attack pattern and apply mutations to the model in order to extract non-intuitive vulnerabilities. In this scenario, every node of the model has a method specified so when the execution traverses through the model, it calls these methods and changes the syntax of the test input. So if the model is mutated afterwards, the nodes are activated in a different order so another injection input is generated and tested against the SUT.

The authors extend this approach further in [6] by using attack patterns as a model but also implement HTTPClient in Java to suit this purpose. In this method, a pattern of cross-site scripting attacks is implemented in YAKINDU Statechart Tools in form of a UML state machine and executed against the SUT. It offers an automatically execution of the attack but also gives the tester the possibility to manually intervene inside the model, thus changing the order of execution but also intermediate values and guiding the execution process through alternate paths with the intention to trigger unknown vulnerability in web applications. If successful, the execution terminates in the final state of the state chart.

In [6] the author discusses the importance of automating security testing in order to be integrated within today's agile software development processes. Moreover, the author also describes common challenges and a potential solution.

3.2 MODEL-BASED SECURITY TESTING FROM SECURITY TEST PURPOSES AND BEHAVIORAL MODELS

3.2.1 Description

This approach, developed by Smartesting within DIAMONDS project, adapts the typical model-based testing process for security testing. This process uses security test patterns and behavioural models as the main drivers for the security test generation process.

The following figure presents the Smartesting process and tool for model-based security testing.

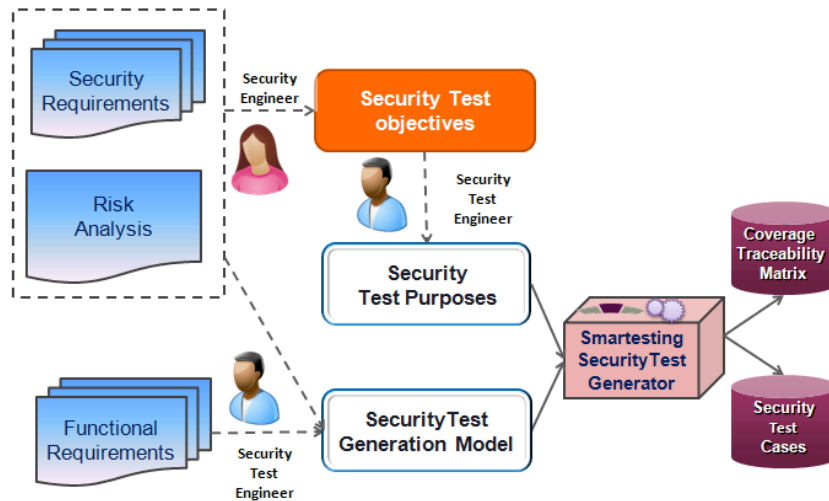


Figure 10: Smartesting process and tool for model-based security testing

Model-based security testing from behavioural models and test purposes is an extension of functional model-based testing (MBT):

- The model for test generation captures the expected behaviour of the system under test (SUT). This model is dedicated for automated generation of security tests, and generally formalizes the security functions of the SUT but also the possible stimuli of an attacker as well as the expected answer of the SUT.
- The test purposes are test selection criteria that define the way to generate tests from the test generation model.

3.2.2 Innovation with respect to the STOA

The main difference with “classical” functional MBT is firstly that the behavioural model may represent stimulations that are not defined in the specification of the SUT. For example, if a security test engineer wants to generate SQL injection test, he or she would represent SQL injection operation inside the test generation model. Secondly, the model-based security testing differs in the way test cases are selected from the behavioural models. In functional MBT, the way is often based on a structural coverage of the model, mixing the coverage of expected behaviour and logical test data. The tests are in general “positive”, aiming to test all the nominal cases and few (or a several) errors cases. In security testing, the goal is really to systematically trying to break (or to bypass) the security functions of the SUT. This search for breaking software security barrier requires to systematically trying possible breakers in a large set of application contexts. The test purpose language goal is to support such test selection criteria.

In this process, the following artefacts are developed and maintained by the Test Engineer:

- The Test Generation Model is a formal artefact developed in a subset of UML/OCL. This is a behavioural modelling fully dedicated for automated test generation.
- The Test Purposes are formal artefacts that define the test selection criteria to be applied on the model to generate the required tests. The test purposes formalize the security testing objectives in an adequate test selection language.

This process is supported by a tool-set based on Smartesting test generation engine, that have been extended with a test purpose language dedicated to security testing to drive test generation from security testing objectives and by a publisher in order to translate generated test cases into executable test scripts on the test bench. This tool-set consists of the following features:

- Support for behavioural modelling activities;
- Support for test purposes design activities;
- Automated test generation;
- Traceability support between test cases and security test objectives.

3.2.3 Tools Support

The details of the tooling are extensively presented in D3.WP3 and D5.WP3 deliverables.

3.2.4 Applications and contributions to case studies

This approach has been experimented on 4 cases studies within the DIAMONDS project:

- Thales HDRM Waveform radio protocols
- SINTEF/NORSE Banking application
- Gemalto Trusted Service Manager
- Itrust LASP (GSM anti-spoofing)

3.3 MODEL INFERENCE ASSISTED EVOLUTIONARY FUZZING

We propose a technique for precise detection of cross site scripting vulnerabilities, using techniques from model inference and evolutionary computing. A prototype of the approach named KameleonFuzz (KF) has been developed and used to test typical web applications.

3.3.1 Description

Given the complexity of modern web based applications, naive black-box fuzzing approaches may not detect deeply nested vulnerabilities. To address the aforementioned problem, we build a model of the SUT by using model inference techniques and guiding the fuzzing process by Evolutionary Algorithm. In this presentation, we focus on cross site scripting (XSS) vulnerability, which is one of the most dangerous web vulnerability (in SANS TOP 25). Figure 11 illustrates a high level overview of the approach. As can be seen in the Figure 11, there are three main steps involved in this approach.

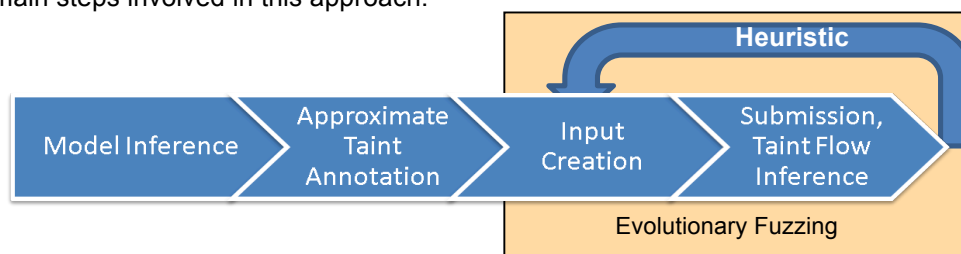



Figure 11: Overview of the proposed approach, showing the main three components

3.3.1.1 Model Inference

The vulnerabilities that exist deep inside the application in terms of data and control flow are difficult to detect by means of input generation. One possible workaround for this problem is to have an idea about how an application consumes its inputs. In other words, what is the state transition of the application in terms of its input/output pairs? Therefore, we make use of model inference techniques to derive the state transition automaton of the SUT. It learns about the control flow and the data flow of the software under test (SUT) by first building a model, and then annotating it with potential reflections. Given a set of interfaces and parameters to connect to the SUT (e.g., authentication credentials), a model is learnt in the form of an Extended Finite State Machine (E-FSM) with instantiated parameters values. This process consists of sending inputs and observing the corresponding outputs. The SUT is a web application and inputs/outputs are an abstraction of HTTP requests/responses. Therefore we make use of a state-aware crawler to learn the model, which is similar to the one proposed in [10].

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 36 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

3.3.1.2 XSS Pattern Detection:

XSS vulnerability allows an attacker to steal sensitive information from a user in web based environment. We focus on two kinds of XSS: reflected and stored/persistent. In order to detect the possibility of XSS, we need to check for the reflection(s) of an input parameter value into a SUT output (block 2 of Figure 11). For this, we use a simple substring matching algorithm with a heuristic to avoid false negatives. Once a match is found, we annotate the transition in which it reflects as a potential sink (T_{dst}) for the input parameter sent in the former transition (T_{src}). This is where the learnt SUT model helps us in detecting the possibility of XSS: we check input parameter values and transitions output to observe if part of the input is present in the next output. In this way, we focus only on the transitions for which this property is observed.

3.3.1.3 Taint Dataflow Inference:


In the case of type-2 XSS, the input is first stored in some intermediate repository e.g., database, and then reflected in subsequent output(s). To precisely detect the presence of XSS, we, therefore, need to precisely track the flow of tainted input from T_{src} to T_{dst} . In a white-box harnessing, this can be achieved by monitoring techniques (e.g., code instrumentation). However, in a black-box harnessing, this is a non-trivial task as we cannot instrument the code, hence the need to infer the taint flow. We could have used simple substring matching algorithm to find if there is a flow of taint input from HTTP requests to HTTP response. However, there are various transformations that may take place at server side and thereby changing the input request. In that case, a simple substring match will miss the taint flow. We, therefore, devise a more complex string matching algorithm, which is based on the work, proposed in [16]. The algorithm computes a distance $\mathbf{TDist}(s1, s2)$ between two strings $s1$ and $s2$ which is a more complex version of edit distance. The string matching is applied on various nodes of the DOM tree of the request/response. For each node of an output parse tree, we compute the string distance \mathbf{TDist} between each tainted substring and the node textual value. Then we keep only the lowest distance score. If this score is lower than a tester defined threshold \mathbf{D} , this node is marked as tainted (i.e. $\mathbf{TDist} < \mathbf{D}$). We obtain such a tree by interacting with browser when connecting with the SUT i.e. we use browser as proxy between our tool and the SUT.

The outcome of the above three steps is an *annotated model*, represented as FSM such that each node corresponds to an abstract state (e.g. logged-in/logged-out) and edges correspond to pages that are accessible from a given page along with the taint information. Taint information consists of parameters that are tainted and the parts of output (i.e. DOM nodes) that are *infected* i.e. tainted by those parameters. With such a representation, it is relatively easy to inspect the SUT to find the possibilities of XSS at certain places.

The SUT may sanitize reflections which may make it impossible to insert any malicious code. Therefore, in order to validate the presence of XSS vulnerability, we need to generate inputs with various malicious cases that trigger the vulnerability. In other words, we need to generate malicious inputs to be executed on the victim's side, including the case when filters are also present to sanitize the inputs at the server side. This is where we introduce the notion of evolutionary fuzzing (rightmost blocks in Figure 11), which is explained in the next section.

3.3.1.4 Evolutionary Fuzzing

Evolutionary fuzzing is a form of fuzzing with ideas taken from evolutionary testing which, in turn, is inspired from the Nature's evolution process [19]. Genetic algorithm (GA) is one among the most widely used evolutionary algorithms that have been used in software testing by generating inputs for the desired properties [10]. We also make use of GA to detect XSS. Figure 12 illustrates the pseudo-code of the evolutionary input generation procedure:

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 37 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

```

1           ▷ Creating the first generation:  $n$  individuals
2 for  $l \in [1..n]$  do
3   Population[ $l$ ] ← generateIndividual(Model,AttackGrammar)
4 end for
5           ▷ Evolving the population
6 repeat
7   for all I in Population do
8     SUT Reset
9     Submit I to SUT
10          ▷ precise taint data flow inference
11     Compute FITNESS(I,Model,0) + VERDICT(I, 0, TreePatterns)
12   end for
13   CROSSOVER:  $f$  fittest individuals according to attack grammar to
    produce  $m$  Children
14   for all C in Children do
15     MUTATE(C,AttackGrammar)
16   end for
17   Population ← ( $n - m$ ) fittest parents +  $m$  Children
18 until stopCondition

```

Figure 12: Pseudo code of Genetic Algorithm

In the following, we explain each part in detail.

Initial Population

Initial population means the initial inputs that will be used as seeds to generate new inputs. After inference, we know the sequence of inputs that will lead us to the state in which the reflection occurs. Therefore, we take that input sequence as it is. Here, we want to elaborate the structure of an individual input. Each input is a HTTP request that consists of several parts (parameters). When the input is at least partly reflected in the output, these parts may be used to form a valid HTML code. Therefore, it is important to supply valid parts with respect to HTML grammar, while taking care of SUT filters. During inference, we use only valid (expected) inputs, whereas during fuzzing, we may not adhere to this assumption. Therefore, each input in the initial population is a sequence of inputs, consisting of the prefix sequence and fuzzed input for the state where the reflection is observed. The fuzzed inputs are produced using the attack grammar, which is explained in the following section.

Attack Grammar

The attack grammar permits generating inputs that manifest a typical attack behaviour. Fuzzed values for reflected input parameters are generated using the attack grammar. It also controls mutation and crossover operators (lines 3, 13, 15 of Algorithm). Attackers would attempt to submit such fuzzed values to the SUT. The following information is included in the grammar:

- What is the targeted structure for this given production rule? (e.g., inside an HTML tag attribute value, outside HTML tag, inside a CSS attribute, . . .)
- Which nodes are mutable? (e.g., OR production rule, bounded repetitions)

In order to create the attack grammar, we use the following knowledge:

- valid HTML grammar production rules [18],
- browser specific string transformations in case of context change,
- known attacks vectors,
- client-side type-1 XSS filters behaviour.

Therefore, while generating populations in GA, we try to adhere to AIG to generate inputs that are close to attacker's behaviour. We represent AIG in BNF and an excerpt of such an AIG is given in Figure 13.

```

1  start = encoding anti_filter charset
      xss_in_structure
2  encoding = ( "plain" | "base64_encode" |
              "php_url_encode" | ... )
3  charset = ( "UTF-8" | "ISO-8859-1" | ... )
4  anti_filter = ( "identity" | "php_addslashes" | ...
                 )
5  xss_in_structure = ( xss_in_url | xss_outside_tag |
                      xss_in_css_background_image | ... )
6  xss_outside_tag = js_script
7  js_script = js_script_start js_payload js_script_end
8  js_script_start = "<script" [0:1] (
                    js_script_start_head ) ">"
9  js_script_start_head = " type=" html_quote
                        "text/javascript" html_quote
10 js_script_end = "</script>"
11 js_payload = (js_payload_1 | js_payload_2 |
                js_payload_3 )
12 js_payload_1 = "location=" js_quote "http://.."
                js_quote ";"
13 js_quote = ("\\" | "'" )
14 ...

```

Figure 13: Input Grammar to generate fuzzed inputs (excerpt)

Based on this grammar, each input is represented as its parse tree, which helps while doing crossover/mutation.

Crossover

As each input in GA is a set of inputs that follow a particular state transition. Crossover can be performed at two levels: 1) between the sets of inputs; 2) between the individual inputs. In the former case, few individual inputs are exchanged, while in the latter case, we change the parts of individual inputs. In the current implementation, we only use case 2. We use two-point crossover, which means that we select a point in both of the parents and exchange parts before and after that point, w.r.t. the attack grammar.

Mutation

Mutation is performed at individual input level, i.e. on the input parameters. As inputs are strings from the attack grammar, our mutation involves three operations to mutate the string: add, replace and delete. These three operations are performed on a particular non-terminal of the attack grammar.

Fitness Function

The fitness function assigns higher scores to inputs that take the execution closer to the vulnerable state. The one we designed is specific to command injection vulnerabilities (XSS being a particular case) [9]. The fitness function is based on certain dimensions that capture the likelihood of an input to trigger XSS. The following table provides several dimensions of our fitness function.

Dimension	Weight
successfully injected character classes	+
tainted nodes in the parse tree	+
singularity	+
number of transitions between source Tsrc and reflection Tdst	+
new page discovered	+
new macro-state discovered	+

If any of the fuzzed parameter value is observed in the output's DOM nodes (neighbours to the node where reflection appears), we detect a XSS vulnerability and the corresponding input can be considered as a test case that triggers XSS vulnerability.

3.3.2 Innovation with respect to the STOA

Existing approaches that combine model/grammar inference and fuzzing, target memory corruption vulnerabilities, whereas KameleonFuzz targets web command injections. Radamsa techniques *grafu* and *permu* [13] infer a grammar from known input sequences then use this new knowledge to generate better inputs. The work in [17] first passively infer a model from network traces, then actively generate and submits inputs to refine this model. Their fitness function is the number of traversed states in the model. KameleonFuzz performs an active model inference from which it produces input sequences that are later executed on the SUT. Fuzzed input values are generated w.r.t. an attack grammar.

Confinement Based Approaches assume that malicious inputs break the structure at a given level (lexical or syntactical). As in [16], we rely on non-syntactical confinement and we use detection policies that are both syntax and taint aware. A key difference is that we used the parser of a browser (e.g., Google Chrome). Thus we have to infer the taint twice. By doing so, we are sure of the validity of found XSS exploits, and our implementation is flexible w.r.t. the browser. The work in [18] relies on non-lexical confinement as a sufficient fault detection measure, which is more efficient than [16], but requires a correctly formed output (which is not obvious on HTML webpages) and is prone to false negatives.

3.3.3 Applications and contributions to case studies

The efficiency of KameleonFuzz has been tested on several **typical web applications** and it has shown promising results by detecting XSS vulnerabilities. The ideas developed in this approach have been applied to **Gemalto Case study**.

3.3.4 Metrics and results

The following table provides results on few real world applications and Gemalto case study.

SUT	# Reflections	# detected XSS
Webgoat	134	6
Gruyere	23	4
Gemalto*	-	0

*While performing Gemalto TSM testing, due to limitations in the access provided, it was not possible to achieve a complete SUT coverage. As a consequence, the reported testing results are related to a limited portion of the SUT.

3.3.5 Tool support

KameleonFuzz is written in Python. It makes use of Selenium browser module for interacting with browser and extracting information.

3.3.6 Exploitation and dissemination

The technique and the experimental results, described in this work, have been presented in international conferences and other technical discussions/meetings provided in D5.WP5 deliverable.

Besides presentations in academic conferences, Grenoble INP has been using the knowledge developed in DIAMONDS, in various courses at master level. There have been efforts to use and extend the ideas in other relevant EU projects that Grenoble INP is involved with. Grenoble INP is also collaborating with industry to re-use the ideas and develop more sophisticated techniques on the basis of aforementioned work.

3.4 STATIC TAINTFLOW ANALYSIS FOR VULNERABILITY DETECTION IN BINARY CODE

Vulnerability analysis, in general, involves the following three steps:

1. Finding some vulnerable spots in the application, e.g., using syntactic or semantic code patterns known from the user experience, known previous attacks, etc.
2. Finding the interfaces for injecting malicious input (tainted data). This is usually derived from some basic knowledge of the functionality of the application. For example, a file reader will probably make use of known library function calls like, ReadFile() or fscanf().
3. Analysing the application (probably at binary level) to find an execution path such that user input can flow to those vulnerable spots.

In a previous deliverable [9], we described our work that addresses the point 1, along with a general description of remaining points. In this present work, we elaborate our work that mainly addresses the last point i.e. discovery of tainted path and few notes on future extension of the whole work leading to inputs generation (test cases) that may trigger some vulnerability of the interest.

3.4.1 Technique Description

As mentioned before, the main objective of this work is to identify tainted dependency paths between pairs of functions calls (IS, VF), where IS is an Input Source and VF a Vulnerable Function. The whole technique involves the following three major components:

1. Creating a call graph based slice;
2. Performing an *intraprocedural* dataflow analysis on each function in a given call graph slice to generate function summaries;
3. Performing an *interprocedural* dataflow analysis on the whole call graph slice for taint propagation and thereby computing the taint flow path.

3.4.1.1 Call graph Slice

To ensure scalability, the solution we propose to retrieve vulnerable execution paths consists in fully analysing only small subset of the program functions. This subset is defined by a slice of the call graph $CG(E, \rightarrow, m)$ where E is set of functions and m is the unique root of the graph. For any two functions s and t , $s \rightarrow t$ iff s calls t . For a given pair (IS, VF), we denote by S_r a slice s.t. r calls (transitively) IS and VF, where r is a common root function of E . It should be noticed that r may not call IS (resp. VF) directly, but through a chain of function calls. If we assume that data can passed through function's arguments and return value, such a slice S_r is a candidate for possible taint flow starting from IS and reaching to VF. Figure 14 illustrates an example of such a slice on a pair (s, t) with r as common root.

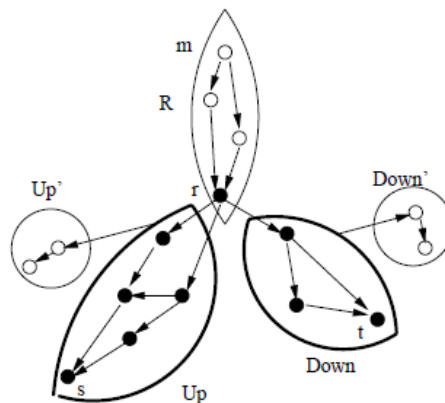



Figure 14: Call graph slice

Where:

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 41 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

- Up (resp. Down) contains the functions that propagate tainted data “upward” (resp. sensitive data downward”) from s to r (resp. from r to t);
- Up’ and Down’ are the sets of functions indirectly involved in data transmission from s to t : they may introduce copies between memory locations.

3.4.1.2 Intraprocedural Dataflow Analysis (IntraDF)

IntraDF is used to calculate the dataflow within a function. This analysis is used on each function f in a slice S_r . The notion of dataflow analysis involves the computation of *data dependence sequences* (DDS), defined as follows:

Let $J = [s_1; s_2; \dots; s_n]$ be an intraprocedural execution sequence, $\mathcal{K} = [s_{k_1}; s_{k_2}; \dots; s_{k_m}]$ is a data dependence sequence (DDS) over J between s_{k_1} and s_{k_m} iff:

- \mathcal{K} is a subsequence of J
- $Def(s_{k_{i+1}}) \cap Use(s_{k_i}) \neq \emptyset$
- $\forall j \in]k_i; k_{i+1}[. Def(s_j) \cap Use(s_{k_{i+1}}) = \emptyset$

A DDS simply expresses the existence of a data-dependency chain between two statements over an execution sequence. As can be noticed, the above defined notion of DDS is strictly applicable to a single function. In order to extend it to interprocedural level, we need to define few symbols as follows:

For a function f we denote by:

- $Arg(f)$, the set of memory locations containing the arguments of f ,
- $*Arg(f)$, the set of memory locations pointed to by an element of $Arg(f)$;
- ret_f the memory location containing the return value of f ,
- $GlobVar(f)$ the global variables accessed in f .

The sets of *inputs* (resp. *outputs*) of f , corresponding to memory locations potentially read (resp. written) by f from the caller’s point of view, are then:

$$Input(f) = Arg(f) \cup *Arg(f) \cup GlobVar(f)$$

$$Output(f) = \{ret_f\} \cup *Arg(f) \cup GlobVar(f)$$


With the above definitions, a DDS at interprocedural level is a sequence of instructions after inlining all the functions and mapping $Input(f)$ and $output(f)$ appropriately at the time of function call and return.

As mentioned above, the main objective of performing intraprocedural dataflow analysis is to create function summaries so that the summaries can be used while performing interprocedural dataflow analysis. The summary of a function f is defined as:

1. $Tainted(f) \subseteq Output(f)$ is the subset of outputs o defined in f at a given statement s_o such that $taint(o, s_o)$
2. $Sensitive(f) \subseteq Input(f)$ is the subset of inputs i defined in f at a given statement s_i such that $sensitive(i, s_i)$
3. $DDep(f) \subseteq Input(f) \times Output(f)$ is the data dependence relation between f ’s inputs and outputs: $(i, o) \in DDS(f)$ iff there exists a DDS over an execution sequence of f between statements s_i and s_o such that s_i defines input i and s_o defines output o .

Summary computation is obtained by using a quite standard forward data-flow analysis based on a classical copy-propagation [12]. Data-flow equations compute, for each statement i and for each memory location x of f , a function $In_i(x)$ (respectively $Out_i(x)$) containing the set of memory locations depending on x before (respectively after) execution of I (i.e. In_i (resp. Out_i): $X \rightarrow 2^X$, where X is a set of memory locations):

$$In_i = \bigcup_{p \in pred(i)} Out_p$$

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 42 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

$$\text{and } Out_i = \mathcal{F}_i(In_i)$$

In these equations the effect of each instruction i is expressed by a transfer function \mathcal{F}_i depending on the nature of statement i . Defining this transfer functions at the binary level is made easier thanks to the basic 3-address structure of the REIL code [20] we use, with only 17 distinct opcodes.

3.4.1.3 Interprocedural Dataflow Analysis (InterDF)

Function summaries are computed for all the functions of each slice S_r . For other functions, e.g. library functions and those appearing in the sets Up' or $Down'$ (see Fig. 4), we use a default summary which over-approximates their side effects:

$$DDep(f) = Input(f) \times Output(f), \text{ Tainted}(f) = \phi, \text{ Sensitive}(f) = \phi;$$

Summaries are computed bottom-up on S_r , starting with the functions in the set Up (from leaf nodes to root r), and then with the functions of the set $Down$ (again from leaf nodes to root r). Within a function f , when an instruction call g is reached, then either g belongs to the current slice (and then its summary has been already computed) or we use a default summary for g . Classical techniques based on fix-point computations over strongly connected components of the call graph could be applied. Finally, in order to reduce the complexity of the dataflow analysis, we take into account global variables only inside the functions directly calling IS and VF (i.e., their immediate predecessors in the call graph). The underlying intuition is that, most of the time, if a global variable v is used inside a program to store a tainted data, then v is usually assigned immediately once this data has been read from IS (and at least in the function which called IS). This choice may miss some vulnerable path, but it can be easily relaxed if necessary.

With the above formalism, we derive the following hypothesis for assert the existence of a vulnerable path: For a given input source IS and a vulnerable function VF, let s_k be a statement defining a memory location x ($x \in Def(s_k)$). The predicate $taint(x, s_k)$, indicating whether x is tainted at statement s_k , holds iff there exists a statement s_i such that:

1. s_i defines a tainted output of IS.
2. there exists a DDS over an execution sequence between s_i and s_k .

Similarly, x is sensitive at statement s_k , denoted as $sensitive(x, s_k)$ iff there exists a statement s_i such that:

1. s_i defines a sensitive input of VF.
2. there exists a DDS over an execution sequence between s_k and s_i .

Finally, a program contains a vulnerable path iff it contains two statements s_1 and s_2 such that:

1. $Def(s_1)$ is tainted at s_1 and $Def(s_2)$ is sensitive at s_2 .
2. there exists a DDS over an execution sequence between s_1 and s_2 .

Figure 15 summarizes the entire technique by providing the pseudo-code of the algorithm.

```

Input: a binary file of a program; an input source IS
Output: a set of vulnerable paths VP

Vulnerable functions (VF) Detection
CALCULATE Set S of Call-graph slices  $S_{IS \leftrightarrow VF}$ , for each VF
for all  $S_r \in S$  do
  PERFORM IntraDF analysis for data dependencies among arguments
  PERFORM IntraDF analysis to trace taintflow (DDS)
  GENERATE functions summaries
  PERFORM InterDF analysis to identify on  $S_r$  the set VP of vulnerable paths between IS and VF
end for
return VP

```

Figure 15: Pseudo-code of the Technique

3.4.2 Innovation with respect to the STOA

There are not so many tools able to perform static taint analysis on realistic applications. A typical example is the Parfait tool [15], which operates at the source level on large C programs and handles some kind of control-flow dependencies. However, the number of available tool reduces further when we consider binary executables analysis. To the best of our knowledge, the only one really close to our proposed work is the LoongChecker tool [8]. This tool also provides an interprocedural summary-based taint analysis on binary executables using the REIL intermediate format. The main difference really lies in the fact that we compute first a call graph slice in order to restrict the sets of functions to be analysed, focusing only on the relevant parts of the code for a given pair (input source, vulnerable function). The experimental results clearly show that this gain in scalability does not prevent us to retrieve the same sets of vulnerabilities. Reps et al. [14] proposed an algorithm for “precise interprocedural program chopping”, but this algorithm relies on specific intermediate program representations (PDG and SDG) which explicit all dependence relations between program statements and predicates. Such structures are therefore expensive to compute, and not appropriate for the analysis of large binary applications.

To summarize, we showed on real examples that this approach scales well, that it retrieves existing vulnerabilities, and that the results produced are small enough to be investigated by hand. This feature can also help developers to patch the bug causing the vulnerability.

3.4.3 Applications and contributions to case studies

A prototype called Light-weight Static Taint Trace (LiSTT) is developed to empirically prove the effectiveness of the approach. We experimented on several *large real world applications*. The approach is also applied on **Metso case study**.

3.4.4 Metrics and results

The following table provides results of experimentation that we performed on various applications including a large binary from Metso.

Application	# functions	# VF	# Callgraph Slices	# Vuln Paths
muPDF	7722	303	47	6
Foxplayer	1074	41	14	6
serenity	559	1	1	1
Metso*	6279	110	23	0

* The Metso experimentations are still not completed and the process is going-on.


3.4.5 Tool support

The LiSTT is implemented as a plugin to BinNavi framework for vulnerability analysis [21]. In order to use BinNavi and performing other preliminary binary analysis, we require IDA Pro [11]. As the interface to interact with BinNavi is Jython/Python, the whole of LiSTT is implemented in Jython/Python programming languages.

3.4.6 Exploitation and dissemination

The technique and the experimental results, described in this work, have been presented in international conferences and other technical discussions/meetings provided in D5.WP5 deliverable.

Besides presentations in academic conferences, Grenoble INP has been using the knowledge developed in DIAMONDS, in various courses at master level. There have been efforts to use and extend the ideas in other relevant EU projects that Grenoble INP is involved with. Grenoble INP is also collaborating with industry to re-use the ideas and develop more sophisticated techniques on the basis of aforementioned work.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 44 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

3.5 ACTIVE FUZZ TESTING

3.5.1 Description

Fuzzing

Bugs are ubiquitous. Essentially all software contains hidden errors either in the form of outright implementation mistakes or subtler design issues. When not spotted during the development or testing processes, they may only manifest themselves when something unexpected imposes stress upon the system. The bug peril affects not only companies building the systems, but also the following layers in the food chain, reaching out to end customers.

Bugs are dangerous for two reasons: first, they cause systems to misbehave which induces all kinds of harm, and second, it costs money to fix the errant behaviour. Software testing tries to address these problems in the pre-release phases of the process. In this report, we will focus on what we consider the single most important conceptual insight a tester should master, and the one which is most often overlooked in practice. Should everyone involved in software testing understand the importance of robustness testing - and fuzzing as the quintessential robustness testing method - we would see fewer exploits, and also fewer and better managed vulnerability patches being pushed to the customers by software vendors.

What does it mean for software to function correctly?

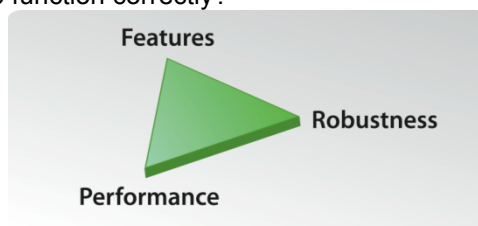


Figure 16: The three different approaches to testing

“Valid functioning” fundamentally comes in three forms, depicted in Figure 16: feature conformance, robustness, and performance. Feature conformance is easy to understand: for all valid use cases, the system should produce the correct outcome. A good system can also handle reasonable amounts of load - that is, performs well in a throughput sense. In the sequel, we simplify things a bit and consider performance as just another feature so that feature and performance testing can be collectively referred to as positive testing. However, something is still missing – robustness and performance are not the same thing.

By robustness we mean the following: Robustness is defined as the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions. That is, a robust system restrains from doing anything undesired when faced with unexpected usage scenarios. These “mis-use-cases” may vary from oddly located mouse clicks or unexpected execution order of actions in a graphical user interface (GUI) to passing broken input through any of the communication channels of the application. Because of the mis-use-case perspective, we also use the term negative testing to refer to robustness testing.

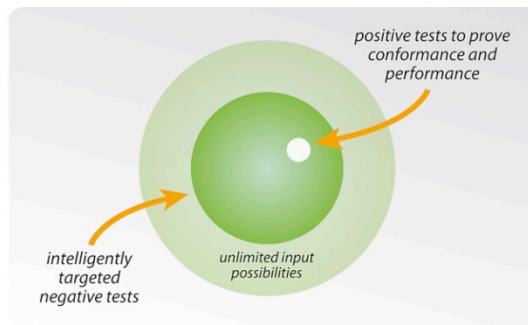


Figure 17: Positive vs. negative testing

The domains of positive and negative testing are illustrated in Figure 17. Project specification typically comes in the form of positive requirements. Sometimes some negative requirements are included, too, but only in a fraction of cases are they comprehensive and thorough. Hence, developers usually have just what we described above: a list of valid-use cases, and very little requirements on mis-use cases.

Another aspect of software development processes is that very typically also the specified and implemented feature sets somewhat differ. Perhaps some of the specified features are not tested and remain missing, and in any case, there are usually loads of “features” outside of what has been agreed upon. Some of them are intentional additions made by the programmers, and some just programming or design mistakes. This phenomenon is visually presented in Figure 18. This discrepancy between specification and implementation is something that further elevates the need for robustness testing since such programmer inventions are typically not documented. If, then, testing is performed according to the original specification, and no robustness testing is performed, one not only misses the unintended behaviour residing in the unspecified domain, but also some parts of the code intentionally produced by the developers.

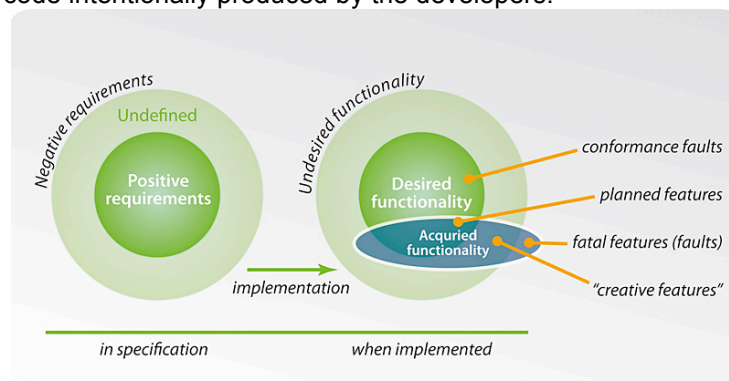


Figure 18: Specified vs. implemented functionality

What happens when negative testing is overlooked?

A shortcoming in testing leads to late discovery of errors – in the worst case, the product has already been shipped to customers. In situations where a released product has serious bugs, post-deployment fixes are needed, and they are very expensive. To think of an extreme example, you can imagine how much it has cost to car manufacturers to execute the recent recall programs to get software issues fixed.

Even when the software is not out yet, late bug discovery may have serious impact on the cost effectiveness of the project. According to Codenomicon studies, Figure 19 schematically presents the relationship between the discovery time of a bug and the cost associated with the fixing related to the bug. While the average costs carry no real figures, it is a commonly accepted fact that the cost of a bug fix increases dramatically with time. This means that thorough testing should be considered loss prevention: to detect and fix issues early in the development process accounts for a great loss that never occurred because of the early discovery.

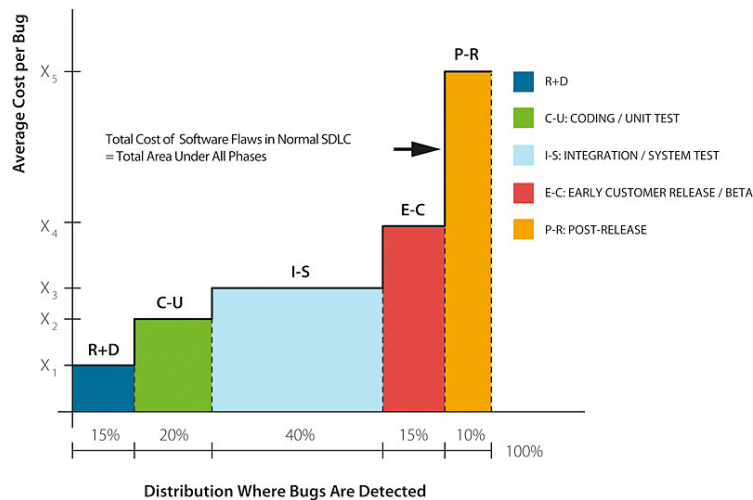


Figure 19: The cost of a bug depending on when it is discovered

The need for negative testing becomes apparent when thinking about products with external interfaces and what typically happens when critical software errors are found after the product is released: very probably message sequences exist that, when sent to the system through its interfaces, cause undefined, unexpected and undesired results. The messages need not even always be sent by a malicious party: it may well be a runtime event relating to some untested program branch, or a unique load situation that is only triggered in real use that wreaks havoc: it may cause anomalous "input" to be fed into the system, and in the worst case, the consequences may be catastrophic. Some of the ways in which unexpected data can be fed into systems are illustrated in Figure 20.

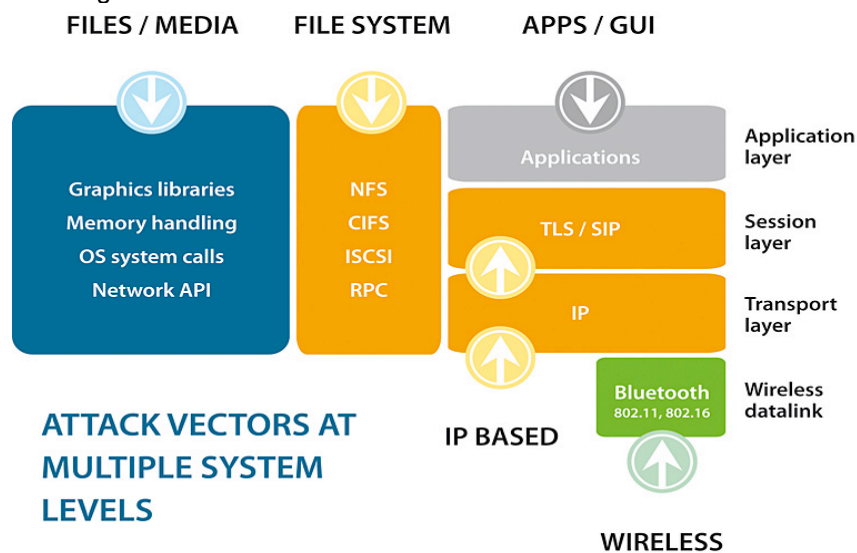


Figure 20: Attack vectors and surfaces

Fuzzing as a method for negative testing

There are many approaches one can take to perform negative testing, but when choosing a tool, one consideration is particularly important: from a hacker's perspective, the input space is infinite so that, when testing software by systematically trying whether it can handle a plethora of malformed input, educated decisions have to be made about which messages to send. The industry leading method for generating sets of effective test sequences is called fuzz testing, or just fuzzing.

The importance of fuzzing stems from the fact it is extremely efficient in exploring the infinite input space in a clever way, and is thus perhaps the only efficient way of finding unintentional security holes. Hence, also hackers invariably try to break into systems using the exact same methods as negative testers, so the builders of software-driven systems should be ahead of hackers and try their methods before deploying a product or a service and exposing it to attackers. An additional advantage fuzzing offers for hackers is that it is a black box testing method, requiring no access to source code that whenever a system is live and accessible through a network interface, hackers can target it by fuzzing.

Automate the Process by Fuzzing

Fuzzing is a technique for intelligently and automatically generating and passing into a target system valid and invalid message sequences to see if the system breaks, and if it does, what it is that makes it break. An important feature of fuzzing is that it requires no knowledge of implementation details of the target system, and when it is so employed, it is thus called a black box testing method. Fuzzers can of course also use some additional information such as the source code of the target, and depending on details, it may then be called grey or white box fuzzing.

By using the term "intelligent" we mean that a good fuzzer does not shoot at random, but rather targets typical programmer errors to increase the rate at which bugs are found. Another benefit from using carefully built test cases is that, whereas it is extremely improbable for a random feed generator to be able to produce something that passes initial validity tests or authentication handshakes, an intelligent fuzzer can be built to do just that and get to test the deeper layers of the target system.

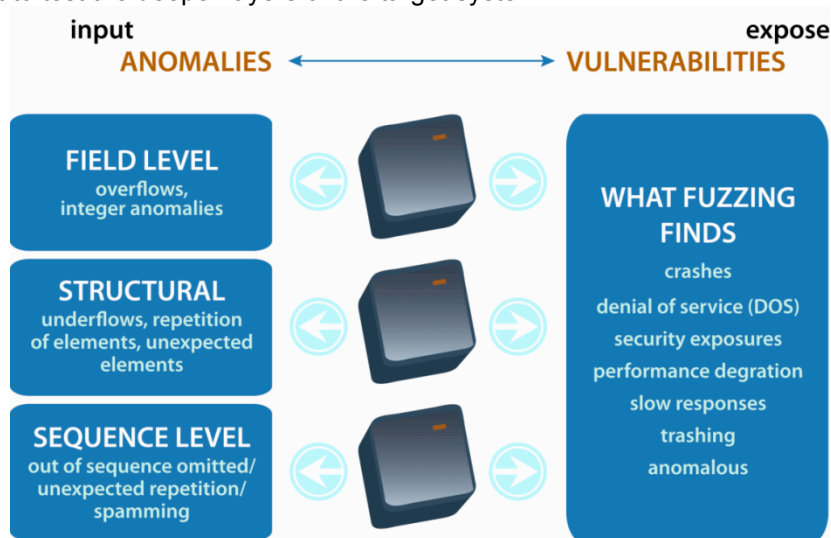



Figure 21: Types of anomalies Defensics produces, and some types of problems it is able to find

What makes fuzz testing so valuable from a security perspective is that it is capable of detecting previously unknown issues that would have been very hard or impossible to discover otherwise – that is, by fuzzing, you can find zero day vulnerabilities.

The Three Categories of Fuzzers

Fuzzers can be categorized into three groups based on how sophisticated they are: random fuzzers, template based fuzzers (sometimes called "mutation fuzzers") and specification based fuzzers. The random variant of fuzzing is becoming more and more of a rarity since it finds bugs at a slower rate and, in practice, fails to discover many issues because of rather strong structure constraints the inputs must fulfil to be actually processed in the target system the first place. Hence, in the sequel, we restrict the analysis to fuzzers based on template data or specifications, and collectively refer to these two types of fuzzing tools as "intelligent fuzzers".

When a fuzzer is specification based (SBF), it includes full knowledge of the protocol specification the test targets, so that it is able to enumerate all or most valid protocol messages and message sequences. This is

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 48 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

not fuzzing as such, but a SBF also includes a mechanism for anomaliing the inputs, that is, making the valid messages into something that resemble true protocol communication, but are a bit wrong in some way or the other. The protocol model and the related anomalizer of an SBF are slow and costly to build, but when a good specification based fuzzer is available, it is a very efficient tool for identifying input sequences that cause problems in the test system.

A template based fuzzer (TBF), on the other hand, means a tool that is capable of capturing real live network traffic and using it to produce similar but mutated message sequences. Just like SBF's, good TFB are efficient, too, and come with the additional benefit that they require no manual specification implementation. The anomalizer is needed, but just like in the case of SBF's, the anomalizers written for different specifications and SUT's share many common properties and methods so that when an anomalizer for a specific situation is available, chances are good that it can be easily modified to address novel test situations, too.

In addition to the fuzzer categorization based on how the protocol model is built, there are yet other division lines in the varied field of fuzzing. The testing tools can, for example, be run on different kinds of hardware. Some fuzzing tools are delivered as remote controllable dedicated appliances that are connected to a network and configured to fuzz the chosen target system. Other fuzzing tools come in the form of software products. Each type of fuzzers has its pros and cons. In situations where the configuration of the fuzzer is straightforward, an appliance might work OK. In other situations, a software fuzzer installed on standard hardware, such as an off-the-shelf laptop, allows for greater flexibility and portability to multiple test setups.

Why Fuzzing Works?

An aspect of intelligent fuzzers that makes them particularly beneficial in testing a protocol aware system is that by making use of the protocol specification or carefully analysed template data, just as intelligent fuzzers do, the tester is able to penetrate deep into the application without being caught by the first input filter or failed handshake procedure. This is also what Makes Fuzzing Work: by not just trying something but bombarding a system with systematically formed messages the tester is capable of testing how individual program states react to different kinds of input sequences, while ignoring messages that would be too broken for the target to accept for processing at all. That is, intelligent fuzzing works because it traverses the infinite input space in such a manner that sequences more probable to cause problems are visited first. Conceptually thinking, random fuzzers are just as good, but they have the immediate practical drawback of taking too long to find anything useful.


Also, a fair share of security vulnerabilities relates to input validation issues. This makes sense: the only mechanism for a user to affect program flow is through one or the other type of input, it is the only way an attacker can make direct use of problems in software. Furthermore, in contrast to most other types of programming errors, input shortcomings in validation are, at least on the borderline, difficult to identify and typically only cursorily addressed in positive testing procedures. Outright program flow errors are much easier to hunt in positive testing settings than subtler issues lurking in the infinity of possible inputs.

Protocol modelling methods used in fuzzing tools

Codonomicon Defensics performs fuzzing, which means that software testing tools have to be capable of first creating valid message structures and message sequences, and then altering these to form nearly-valid messages that systematically anomalize some parts of the information exchange to test the target system for robustness. This section describes some methods and techniques both open source (Peach as an example) and commercial (Defensics as an example) fuzzing tools use for formally specifying messages and message sequences in the context of communication protocols.

A communication protocol is a formal system defining digital message formats and the rules describing the exchange of the messages. Thus, protocol modelling essentially consists of building a formal syntax that enables to create valid messages or confirm the validity of messages created by others, and also to create and validate sequences consisting of several messages.

The actual models that fuzzing tools use are typically variants of industry standard protocol description languages, so we will start by explaining those standards first.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 49 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

EXAMPLES OF PROTOCOL MODELING TECHNIQUES

Augmented Backus-Naur Form

The Backus-Naur Form is a popular method of describing individual messages. As such, it is insufficient for describing message sequences. An extension to BNF which can also contain message exchange descriptions is the Augmented BNF, or ABNF.

Below is a snippet of (standard) ABNF

(from http://en.wikipedia.org/wiki/Augmented_Backus%E2%80%93Naur_Form):

```
postal-address = name-part street zip-part

name-part     = *(personal-part SP) last-name [SP suffix] CRLF
name-part     =/ personal-part CRLF

personal-part = first-name / (initial ".")
first-name    = *ALPHA
initial       = ALPHA
last-name     = *ALPHA
suffix       = ("Jr." / "Sr." / 1*"I" / "v" / "x")

street        = [apt SP] house-num SP street-name CRLF
apt           = 1*4DIGIT
house-num     = 1*8 (DIGIT / ALPHA)
street-name   = 1*VCHAR

zip-part      = town-name "," SP state 1*2SP zip-code CRLF
town-name     = 1*(ALPHA / SP)
state        = 2ALPHA
zip-code      = 5DIGIT ["-" 4DIGIT]
```

ASN.1

Another common format used in test automation such as fuzzing is the standard Abstract Syntax Notation One (ASN.1). It is mostly used for protocols whose specifications are written using it. ASN.1 is a method for describing the collection of individual messages in a protocol, not the message flow.

Example of ASN.1 (from <http://portal.etsi.org/mbs/languages/asn.1/asn1abstract.htm>):

```
ExampleProtocol DEFINITIONS ::=
BEGIN
  ExampleProtocolMessage ::= CHOICE {
    rlcGeneralBroadcastInformation  RlcGeneralBroadcastInformation,
    rlcDownlinkPhyModeChange       RlcDownlinkPhyModeChange,
    rlcDownlinkPhyModeChangeAck    RlcDownlinkPhyModeChangeAck,
    rlcFrequencyList               RlcFrequencyList,
    rlcConnectionAdditionSetup     RlcConnectionAdditionSetup,
    rlcConnectionAdditionAck       RlcConnectionAdditionAck }


  RlcGeneralBroadcastInformation
    ::= SEQUENCE {
      duplexMode           DuplexMode,
      frameOffset          FrameOffset,
      uplinkPowerMaxRangingStart UplinkPowerMax,
      infoText             InfoText }

  DuplexMode ::= ENUMERATED { fdd(0), tdd(1) }
  FrameOffset ::= INTEGER (0 | 8..20)
  UplinkPowerMax ::= INTEGER (10..20)
  InfoText ::= IA5String (SIZE (0..128))
  RlcFrequencyList ::= SEQUENCE (SIZE(32)) OF PairOfCarrierFrequencies
  PairOfCarrierFrequencies
    ::= SEQUENCE {
      uplinkCarrierFrequency CarrierFrequency,
      downlinkCarrierFrequency CarrierFrequency }

  CarrierFrequency ::= INTEGER (0..130000)
END
```

Another example from the X.509 standard that is used for specifying certain cryptographic-relates message exchange rules:

```
TBSCertificate ::= SEQUENCE {
  version          [0] Version DEFAULT v1,
  serialNumber     CertificateSerialNumber,
  signature        AlgorithmIdentifier,
  issuer           Name,
```

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 50 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

```

validity          Validity,
subject           Name,
subjectPublicKeyInfo SubjectPublicKeyInfo,
issuerUniqueID [1] IMPLICIT UniqueIdentifier OPTIONAL,
                  -- If present, version MUST be v2 or v3
subjectUniqueID [2] IMPLICIT UniqueIdentifier OPTIONAL,
                  -- If present, version MUST be v2 or v3
extensions        [3] Extensions OPTIONAL
                  -- If present, version MUST be v3 -- }

```

XML/Schema

For testing any interfaces that use the XML specification format, the most common language that fuzzers need to support is the XML/Schema which is a method for the description of valid XML files.

Example of XML/Schema (http://www.w3schools.com/Schema/schema_example.asp):

```

<xs:element name="shipto">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="address" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="country" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

PCAP with PDML

Fuzzing tools can also use Packet Details Markup Language (PDML) dissection of PCAP recordings to create models directly from captured messages. The PCAP + PDML are typically used in parallel as PDML alone is not enough to specify messages for test generation purposes. PDML is originally intended for describing the structure of dissected packets.

Example of PDML:

```

<pdml>
  <packet>
    ...
    <proto name="IP" pos="15" showname="IPv4 (Internet Protocol version 4)"
size="20">
      <field name="verhlen" pos="15" show="45" showname="Version and header Length"
size="1" value="45">
        <field mask="F0" name="ver" pos="15" show="4" showname="Version" size="0"
value="4" />
        <field mask="0F" name="hlen" pos="15" show="5" showname="Header length"
size="0" value="5" />
      </field>
      ...
    </packet>
  </packet>
  ...
</pdml>

```

The PDML specification with samples is available here:

<http://gd4.tuwien.ac.at/.vhost/analyzer.polito.it/30alpha/docs/dissectors/PDMLSpec.htm>

PROTOCOL MODELS IN PEACH FUZZING FRAMEWORK

Peach is an open source fuzzing framework used to build fuzzing tools.

Peach is available from: <http://peachfuzzer.com/>

Peach PIT files


Peach uses XML Schema augmented with BNF and ASN.1 functionality.

Example data templates for text protocols (from <http://peachfuzzer.com/HowDoI/>):

```

<!-- Create a simple data template containing a single string -->
<DataModel name="HelloWorldTemplate">

```

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 51 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

```

    <String value="Hello World!" />
  </DataModel>

  <StateModel name="State" initialState="State1" >
    <State name="State1" >
      <Action type="output" >
        <DataModel ref="HelloWorldTemplate"/>
      </Action>
    </State>
  </StateModel>

```

ASN.1 example ([from http://peachfuzzer.com/HowDoI/](http://peachfuzzer.com/HowDoI/)):

```

<DataModel name="Asn1Ber">

  <!-- define a ber encoded string -->
  <String type="char" value="Hello World!">
    <Transformer class="asn1.BerEncodeOctetString" />
  </String>

  <!-- define a ber encoded integer -->
  <Number size="16" signed="true">
    <Transformer class="asn1.BerEncodeInteger" />
  </Number>

</DataModel>

```

CODENOMICON DEFENSICS SUPPORTED TECHNIQUES

Supported formats:

- RAW ASCII
- BNF
- ABNF
- ASN.1
- XML/Schema
- PCAP
- PDML

Internally used, proprietary formats:

- EBNF
- XML Sequences

We will first explain the Codenomicon proprietary formats/languages, and then show how a user can use these in the tool.

Extended Backus-Naur form with Java extensions

The core modelling technique at Codenomicon is an in-house extended BNF variant that allows for message exchange descriptions. The technique is called Extended BNF, or EBNF.


An example of Codenomicon proprietary Extended BNF (EBNF):

```

TLS ClientHello:
client-hello = Handshake: {
  msg_type: { HandshakeType.client_hello },
  body: { @sid-len @cs-len @cm-len (
    !hs:client-version protocol-version
    !hs:client-random Random
    .session_id_length: !sid-len:target uint8
    .session_id: !sid-len:source (!hs:client-session-id SessionID)
    .cipher_suites_length: !cs-len:target uint16
    !cs-len:source !hs:client-cipher-suite !cipher-suite:cipher-type cipher-suites
    .compression_methods_length: !cm-len:target uint8
    !cm-len:source compression-methods
    # RFC4366: TLS Extensions
    .extensions: ()
  ) }
}

```

We use extended BNF to model the syntax of messages in both binary and textual protocols. Message sequence-level behaviour is also modelled using BNF.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 52 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

The way Extended BNF works on the message sequence level is that it uses rules to append the model with callbacks to Java code. Rules are used to:

- Perform I/O operations like connect, send, and receive, on message sequence level.
- Calculate fields like lengths, checksums, sequence numbers, inside protocol messages.

User sequences

In addition to providing the end user with lots of ready-made test cases - that is, anomalized messages and message sequences - the users of several Codenomicon test tools may also specify the used message sequences and/or message content themselves. For this purpose we use an in-house developed XML based sequence file method. A sequence file may look like the following (the extract has been copied from Codenomicon SIP test suite):

```
<sequences>
  <!-- SIP dialog definitions -->
  <sequence name="used-dialog" setting="user-sequence-file">
    <description name="PUBLISH request">A sequence sending a PUBLISH request
      and expecting a success response.</description>

    <send name="publish-request" type="sip-message" description="PUBLISH request">
      <content file="sip-publish.txx" format="text"/>
      ...
      <store attribute="call-id">...</store>
      ...
    </send>

    <recv name="publish-response-ok" type="sip-message" description="OK response">
      ...
      <match attribute="call-id">!sip:callId $publish-request:call-id</match>
      ...
    </recv>
  </sequence>
</sequences>
```

The actual content of the send message "publish-request" is specified in a separate file as raw send data. The sequence file specifies that SIP header "call-id" must match the corresponding header line in the received message. Note that the real sequence specifications contains a lot more details than included here as the SIP protocol is rather complex.

Model Editing Capability in Defensics

As seen in the Defensics screenshot below, the users can launch editors to edit both the sequence models and the message structures. The models are edited as text.

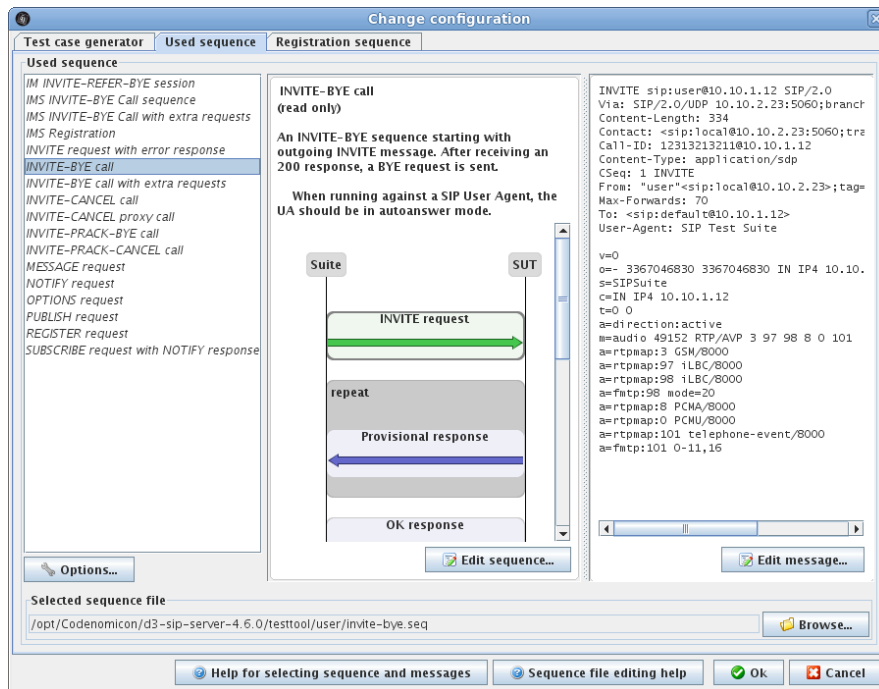


Figure 22: Editor for sequence models and message structures

Conclusion

Section 3.5.1 illustrated that when testing software to ensure its quality and security, it is essential to consider all three aspects of testing:

- Feature testing
- Performance testing
- Robustness testing

As a further incentive to test properly, we demonstrated that sound testing also has positive cost-benefit effects due to the fact that an early discovery of bugs enables them to be fixed quickly, without additional costs.

Moreover, we introduced fuzzing as the standard robustness testing methods and summarized the three categories of fuzzers:

- Random fuzzers
- Model based fuzzers
- Specification based fuzzers

Remembering the messages displayed in Figure 16 and Figure 17, we conclude this section by emphasizing that to achieve software security, it is essential to harness the power of robustness testing to its full potential, and never look back again.

Fuzz test generation strategies

Fuzzing is a technique for intelligently and automatically generating and passing into a target system valid and invalid message sequences to see if the system breaks. A practical example of fuzzing would be to send malformed HTTP requests to a web server, or create malformed Word document files for viewing on a word processing application. The web server example is graphically illustrated in the figure below.

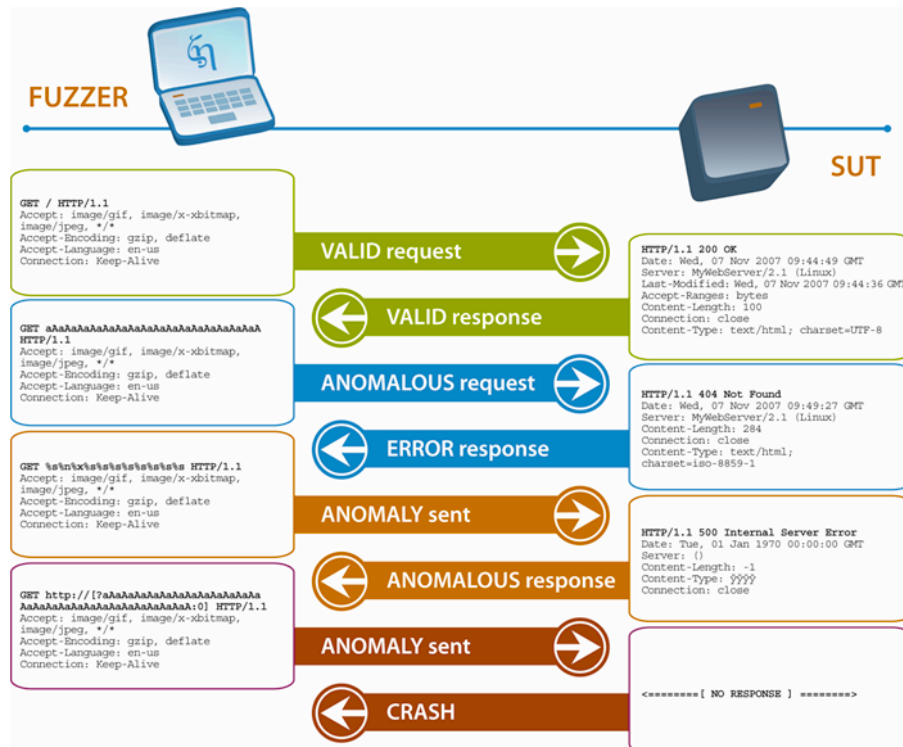


Figure 23: Web server example for fuzzing

Fuzzing has proven to be the most efficient method for finding previously unknown vulnerabilities in software. This is due to the nature of software creation processes: invariably, irrespective of the original specification, features end up in the resulting implementation without no-one knowing where they are there. These features may be not be harmful, and are sometimes even useful. However, a majority of them cripple the robustness of the software and sometimes manifest themselves as exploitable vulnerabilities that hackers can make use of to compromise systems. This gap between specification and implementation is illustrated in the Figure 18.

As stated earlier, fuzzing comprises transmitting input sequences to a target system. This is performed as part of the following process:

1. Create protocol model (if possible)
2. Create test cases from the model or from templates
3. Analyse test results by monitoring the test target


In the context of model-based fuzzing, a model can always be created - if nothing else, it's just a simple model that produces purely random sequences of bits.

The test case generation as part of the fuzzing process is based on the acquired protocol model. How to do this depends on various factors, including the following:

- Is there a "grammar" (specification) available that specifies which kinds of inputs are valid and which are not?
- If not, is it possible to capture some kinds of example sequences as templates?
- Can the creator of the fuzzer afford the time to implement a system that can produce valid message sequences to an extent that represents the space of all input sequences sufficiently well?

Specification based interface models

If there is a specification and enough time available, the most efficient fuzzer can be created by modelling the specification of the communication interface, and use that model for creating valid message sequences. The test generation is then about insert anomalies into the sequences. These anomalies are message elements that are known to cause problems in software. That is, the valid input sequence is broken in a way so

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 55 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

as to be as likely to create problems in the target system as possible. This approach that uses a protocol specification is called specification based protocol fuzzing.

Specification based fuzzing tools come in two flavours:

- Either the model has been pre-built into the tool so that the user just needs to connect it to the target system and start testing
- Or, alternatively, the "test tools" is just a framework that facilitates the process of creating the protocol model, but contains no model as such

Template based models

If the input model cannot be created from a specification, one can in some cases capture example sequences and, if their structure can be analysed to a sufficient degree, modify the examples with inserted anomalies and then use the resulting sequences as test cases. The templates used to build the model may, depending on the situation, be flat binary data or, for example ASCII character streams, and the model may be constructed with no relation to the underlying protocol or file specification. Alternatively, if an attempt to actually model the protocol is wished, more advanced algorithms exist that try and understand the input structure on a more sophisticated level.

Random model

If all else fails, test cases can also be created partially or fully randomly. Even though this approach contains the least amount of insight into the target system, even random test cases are known to often trigger bugs in software.

Test generation techniques

When the model has been built, one needs next to create the test cases. This may be performed in one of the following ways:

- **Online:** Create the test cases in real time as testing proceeds, basing new test cases on what has been done previously
- **Offline:** Create static tests that are later injected to test target. The "static tests" may be files, pre-formatted message sequences or other types of input.

Conclusions


According to studies by Codenomicon, all types of fuzzing discover new bugs and they all have their time and place. The more input awareness one can build into a fuzzer, the more efficient it is likely to be. Therefore, whenever sophisticated specification based fuzzers are available for a specific purpose, they are the safest bet to use.

It is a difficult and time consuming task to build good specification based fuzzers so that if a fuzzer is needed for a novel purpose, with no off-the-shelf tool available, the decision boils down to whether it makes sense to build one from scratch, possibly using a fuzzing framework, or deploy a template based fuzzer instead.

In some situations, also test execution speed may be critical. In these situations, specification based fuzzers also shine as the test cases can be pre-built according to the specification, and then efficiently injected into the target system as the test proceeds. This partly applies to template based and random fuzzing, too. However, in the template-based approach, is often beneficial to first do some work with finding good templates. Random fuzzing suffers from a need to have a huge deal of test cases for even a moderate coverage of the input space.

3.5.2 Innovation with respect to the STOA

Advances were made in combining the different fuzzing methods. Codenomicon is largely a manufacturer of generational fuzzers. The model based tools achieve good coverage of test system fast, but the implementation is slower. To compensate the capture based fuzzing was enhanced with modelling. Capture based fuzzer can utilize the open source community provided dissectors to identify elements and fields (for example length fields can be detected, data types can be deduced). Codenomicon implemented additional value by

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 56 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

allowing to enhance the community dissector data with own dissection information. Also it was made possible to add dynamic rules to the capture based sequences. Dynamic rules can calculate checksums or sequence identifiers, which need to be correct for the testing to achieve deeper state machine penetration in the system under test. Also it was made possible to define totally new rules for the capture tool, even for customers. This combination of model based and template based fuzzing has achieved good results and allowed to implement solutions much faster.

Template based fuzzing was added also to model based tools. File formats or captures can be read into the model, enhancing the data that is used in testing which can achieve better testing coverage. For example a protocol requires a configuration identifier to be in place, and that kind of value can be identified from the template to the model, which makes tools easier to use and when correctly configured more effective.

Fuzzing types were also combined in a way that user is able to choose to do mutational test cases. Default test cases are based on the model and intelligently breaking the value and structure assumptions. To enhance testing, user can then choose to combine test cases and create additional test cases by mutations of the default test cases.

These combinations have greatly enhanced the testing effectiveness, and have uncovered issues in already well tested targets.

3.5.3 Applications and contributions to case studies

Telecommunication case study and Industrial Automation case study (Please refer to D5.WP1).

3.5.4 Metrics and results

Results are reported in Telecommunication and Industrial Automation Case study reports in D5.WP1.

3.5.5 Tool support

The above described advances are available in the test tools created on the latest version of the platform (Please refer to D5.WP3).

In the Diamonds project the support for WLAN, Bluetooth low energy and GSM radio testing was implemented. WLAN and Bluetooth support is adaptation of the radio modules to the testing platform. The commercial solutions on the area already place a big part of the radio protocol communication to be done on the PC host, and replacing that was pretty straightforward. Models can be implemented and fuzzing can be done on these radio protocols.

Cellular radio was studied and a terminal SMS test solution was implemented. Cellular radio is much harder environment, the radio protocols, due to the licensed spectrum, are implemented in the HW and ASICs. A system was found where the GSM base station stack is implemented on a PC. This allowed to implement terminal testing solution, where the radio protocols can be fuzzed. First implementation is SMS, due to the large usage base, direct billing consequences and utilization in different kinds of M2M communications. Cellular area needs focused implementation effort due to the complexity, and the testing of the base stations and using 3G and LTE radios remains a study area.

3.5.6 Exploitation and dissemination

All exploitation results are under Fast Exploitation provided in D5.WP5 deliverable.

3.6 MODEL-BASED BEHAVIOURAL FUZZING

“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.”[1] 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 traditional approach of data fuzzing is generating invalid input data the SUT is fed with. This is a successful applied method to find vulnerabilities [3], [4]. Behavioural 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.

Behavioural 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 [9], [11]. Hence mutation testing is a white box approach. In contrast (behavioural) 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 behavioural 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 nice real-world example is given in [12] 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 behavioural fuzzing would change the number of host messages sent to the web server. Only an invalid number of host messages generated by behavioural fuzzing can reveal this denial of service vulnerability.

3.6.1 Technique description

3.6.1.1 Model-based Behavioural Fuzzing of UML Sequence Diagrams

We use as a starting point UML sequence diagram that represent valid message sequences. Test cases are generated by **fuzzing one or more valid sequences**. This concrete fuzzing of behaviour is realized by changing the order and appearance of messages in two ways:

- **By rearranging messages directly.** This enables straight-lined sequences to be fuzzed. Such fuzzing operators modify either a single message, e.g. by removing or repeating it or by replacing it by another, or the sequence of messages, e.g. by moving a message from its original position to another, or rotating a set of subsequent messages by stepwise moving the last message of the subsequence to the beginning of the subsequence. This is depicted by Figure 24, it shows the application of the fuzzing operator *Remove Message* to a valid sequence diagram in order to remove the authenticating *logon* message.
- **By utilizing control structures of UML 2.x sequence diagrams**, such as combined fragments, guards, constraints and invariants. This allows more sophisticated behavioural fuzzing that avoids less efficient random fuzzing. Fuzzing operators are for example negate interaction constraint that negate a guard of an interaction operand, change bounds of loop or disintegrate combined fragment.

By applying one or more fuzzing operators to a valid sequence, **invalid sequences (= behavioural fuzzing test cases)** are generated.

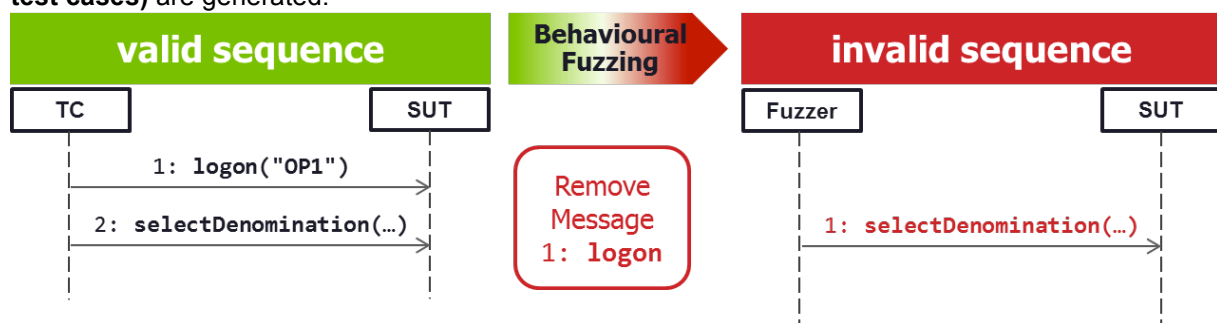



Figure 24: Application of a behavioural fuzzing operator in order to generate an invalid sequence

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 58 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

An overview of the different behavioural fuzzing operators for UML sequence diagrams and a detailed description as well as the test generation approach can be found in the DIAMONDS project deliverable D2.WP2.

3.6.1.2 Behavioural Fuzzing for Authentication Bypass Vulnerabilities

The complexity of the space of test cases that can be generated is too large. Executing all these test cases is impossible. Therefore, an appropriate selection technique is required in order to generate and execute such test cases that have the largest chance to reveal a vulnerability. The information how to select the test cases can be assisted by a risk analysis. As a result from the risk analysis, a set of vulnerabilities may result that should be tested for. This could be an authentication bypass vulnerability where a system exposes protected resources and functionality without authentication checks. Such a vulnerability can be addressed by behavioural fuzzing.

By annotating a sequence diagram with the information about authentication and deauthentication messages and the messages that shall only be executed with authentication, behavioural fuzzing is able to generate a set of test cases that test exactly for authentication bypass vulnerability. For the annotation of a sequence diagram, we adopted UMLsec's annotation for role-based access control. UMLsec's role base access control stereotype is applied to UML activity diagrams and has three attributes:

- **protected** whose values contains the protected resources that are states of the *rbac* annotated activity diagram.
- **role** contains tuples of an actor (of the annotated activity diagram) and a role where the actor is assigned to the role.
- **right** describes which roles have permissions to access a protected resource by corresponding tuples.

We adopted this stereotype such that it can be applied to UML sequence diagrams and has an additional attribute *authentication* that contains tuples of authentication and deauthentication messages, generally login and logout. This is illustrated in Figure 25. It shows a sequence diagram consisting of four messages. The attribute describe that an *operator* has the role *money counter*. The role are *rights* assigned to access the protected resources *configure* and *countMoney*. The new attribute *authentication* identifies the authentication message *login* and the deauthentication message *logout*.

Using this information, behavioural fuzzing can generate explicitly test cases that test for an authentication bypass vulnerability by an invalid message sequence:

- The fuzzing operator *Move Message* can now only move the login and logout messages. Login messages can be moved stepwise closer to the logout message to test if the messages appearing after the login can be successfully executed without authentication. Accordingly, the logout message can be moved stepwise closer to the login message to test if the logout is successful and no operations can be executed after a logout.
- *Remove Message* may consider only the login message in order to test if messages that need authentication can be performed without.
- *Repeat Message* may only repeat the login and logout message in order to check if the authentication state remains unchanged by the repeated message.

Using this mechanism, model-based behavioural fuzzing can be applied in order to test for certain vulnerabilities, e.g. authentication bypass vulnerabilities. At the same time, it reduces the number of test cases to be executed. More details on this approach and a comparison of the number of test cases with and without this approach can be found in the DIAMONDS project deliverable D3.WP2.

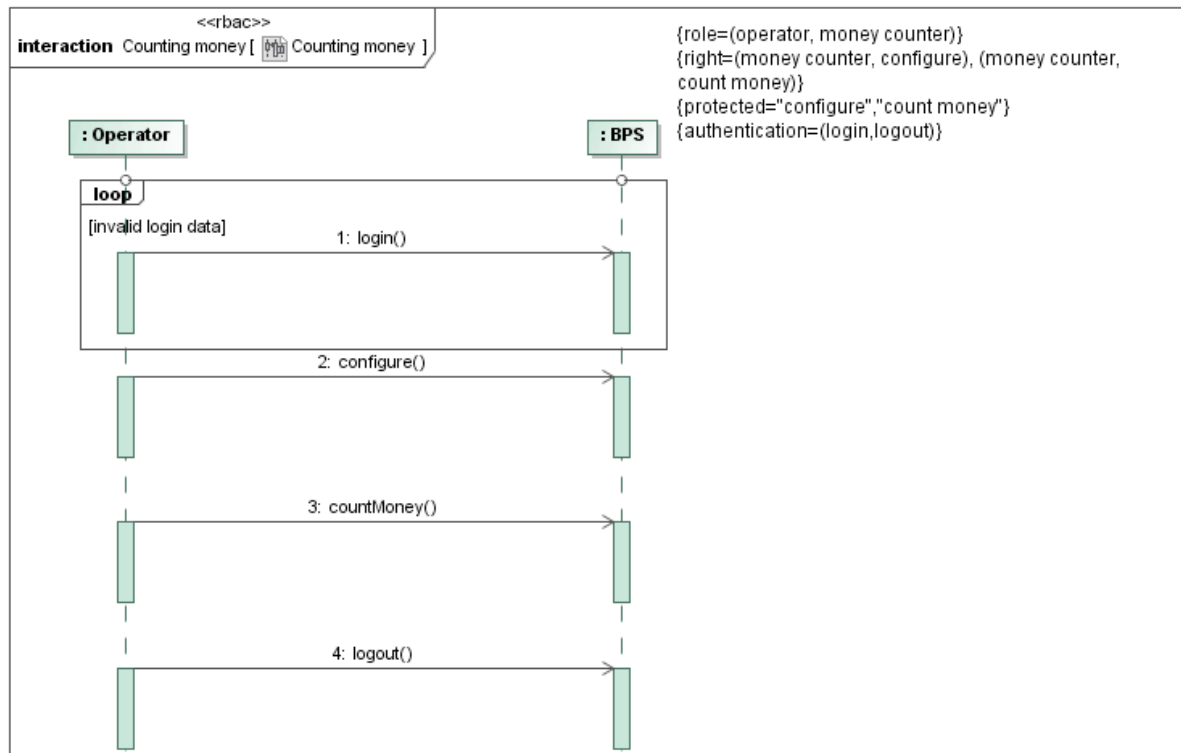


Figure 25: Sequence diagram with extended RBAC stereotype

3.6.1.3 Online Model-Based Behavioural Fuzzing

Online model-based behavioural fuzzing is an approach to make the test execution for behavioural fuzz testing more efficient by


- generating test cases at runtime instead of before execution,
- focusing on interesting regions of a message sequence based on a previously conducted risk analysis, and
- reducing the test space by integrating already retrieved test results in the test generation process.

Online Test Case Generation

Online test case generation means that test generation and execution are taking place at the same time having a mutual impact. The idea is to execute test cases as long as the SUT is healthy. Obviously, the SUT does not need to be restarted as long as it behaves normally. Thus, long startup times of the SUT can be avoided. Test cases are generated on demand and are submitted message-wise to the SUT as long as it is healthy. If the SUT is not healthy after submitting a message, probably a weakness was found. In this case, the submitted message sequences are persisted for later analysis. If the SUT behaves normally after submitting the message sequence of the whole test case, no weakness was found and the submitted test case does not need to be persisted. The approach of online generation avoids generating and saving all the behavioural fuzz test cases. Instead, only message sequences that actually revealed weaknesses are saved for later analysis.

Integrating Previous Test Results

After sending a (possibly invalid) message from the test execution environment to the SUT, its impact on the SUT is determined. If the SUT behaves anomalously, a weakness in the SUT was found. Generated test cases may contain message subsequences of other test cases. If such a subsequence already revealed a vulnerability, other test cases containing this subsequence do not need to be executed because they wouldn't find any other weakness. This is realized by keeping the message sequence of each test case up

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 60 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

to the message that revealed a weakness. If a new test case is generated, it is compared with these message sequences in order to avoid generating as part of a new test case. This improves efficiency because this also avoids restarting the SUT when no new weakness was found.

Focusing on Message Subsequences

Behavioural fuzzing of a complete sequence diagram is mostly not useful. Some parts of the sequence may contain preamble and postamble steps which set the SUT into the correct mode and are not part of the test itself. Moreover, the focus of security tests is more narrowed to potential threats and vulnerabilities that result from a previously conducted risk analysis. For example, a result from a risk analysis could be that the SUT might be vulnerable in a certain state. Therefore, testing for that vulnerability is only useful if the SUT is in that state. By defining regions to be fuzzed, two benefits can emerge:

- It may reduce test execution time by defining regions that avoid time-consuming messages. While this reduces the execution time of a test case, some weaknesses would be missed when excluding some messages. Therefore, the definition of regions has to be done by experts.
- Using regions has a decreasing impact on the number of elements fuzzing operators can be applied to. Thus, the overall number of test cases may be reduced.

The annotation of sequence diagrams with information about role-based access control is also an example how such a region can be defined.

3.6.2 Innovation with respect to the STOA

While behavioural fuzzing isn't a completely new approach, it is rarely elaborated. It first mentioned by Kaksonen et al. [19] during the PROTOS project (2001): "*In fault injection [i.e. fuzzing] mutations can be applied to the syntax of individual protocol messages as well as the order and type of messages exchanged.*" Thus, Kaksonen et al. did not restrict fuzzing to generating invalid input data but define it also what we denote by behavioural fuzzing. Becket et al. [22] applied behavioural fuzzing to state machines by inserting, repeating and dropping messages based on the behavioural model. SNOOZE [18], a tool for building for network protocol fuzzers uses state machines for protocol description and provides primitives for retrieving invalid messages depending on a state and thus, enables developing behavioural fuzzers. Kitagawa et al. [12] showed the effectiveness of behavioural fuzzing by finding vulnerabilities, e.g. in Apache web server that could not be found by data fuzzing, but did not describe their test generation method.

Behavioural Fuzzing was extended to UML sequence diagrams by providing a rich set of behavioural fuzzing operators explicitly made for sequence diagrams. These fuzzing operators take advantage from the structures of sequence diagrams, e.g. combined fragments. A concrete method for generating test bases by applying fuzzing operators was developed. The approach enables reusing of functional test cases for security testing. A method how test case generation may benefit from risk assessment results by augmenting the model with security-related annotations was presented. This helps to focus on certain security aspects while reducing the number of test cases.


Online model-based behavioural fuzzing combines the model-based behavioural fuzzing approach with the risk analysis and online test case generation in order to improve the efficiency of the behavioural fuzzing approach.

3.6.3 Applications and contributions to case studies

Model-based behavioural fuzzing was applied to Giesecke & Devrient case study. For that purpose, functional test cases delivered by Giesecke & Devrient were used as a starting point. On the basis of these functional test cases, an appropriate test case based on a risk analysis was selected and modelled as UML sequence diagram. A prototype of a test case generator for UML sequence diagram generated a set of test cases. From this, 30 test cases were manually selected according to the risk analysis. More details can be found in the DIAMONDS project deliverable D5.WP1.

3.6.4 Metrics and results

Model-based behavioural fuzzing was applied to the Giesecke & Devrient case study. For that purpose, 30 test cases were initially generated and executed. However, the execution of such a test case on the case

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 61 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

study takes considerable time. Hence, approaches to optimize the efficiency of the approach were investigated. This led to two results, the test selection method for authentication bypass vulnerabilities based on an extension of the UMLsec profile for role-based access control stereotypes for UML sequence diagrams. On the other hand, the online approach results from this challenge that integrates test generation, execution and testing for authentication bypass vulnerabilities and thus, improves the efficiency of the model-based behavioural fuzzing approach when applied to complex systems.

3.6.5 Tool support

The test case generator expects as input the XMI format for UML sequence diagrams and thus, supports a lot of modelling tools, e.g. MagicDraw and Papyrus. The immediate output of the test case generator are also UML sequence diagrams as test cases. From these, executable TTCN-3 code is generated by a code generator that allows execution of test cases by every test execution tool that is conform to the TTCN-3 standard, e.g. TTworkbench and Telelogic Tau Tester.

3.6.6 Exploitation and dissemination

A prototype of a test case generator was developed as an Eclipse plugin and extended for the online approach. FOKUS will provide services for introducing fuzzing and developing product-specific fuzzing heuristics based on experiences made with model-based behavioural fuzzing.

FOKUS made publications on several conferences and workshops, among them on the MBT user conference, the SAM workshop and the SECTEST workshop provided in D5.WP5 deliverable.

3.7 SOURCE CODE CHANGE BASED TEST CASE SELECTION

In development of software intensive systems testing is still one of the most important security and quality assurance method. Software is developed incrementally, i.e., new software is developed by extending or modifying existing software. In practice, the difference between a new software version and the version upon which it is built is small compared to the whole size of the software. This means that quite likely most of the test cases are such that they do not go through a modified software location. In other words, those test cases will execute exactly the same sequence of source code level instruction for the new version and the old version. In automatic source code change based test case selection we try to utilize this feature by automatically selecting for re-testing only those test cases that will go through a modified software location.


3.7.1 Technique description

The technique is based on work by Rothermel and Harrold¹³ and Ball¹⁴. There are three phases in the analysis. In the first phase control flow graphs of the new and the old version are created. In the second phase edges of the control flow graph of the old version such that the corresponding edge of the control flow graph of the new version will lead to a modified software location are identified. These edges are called dirty. In the final phase the test cases that go through a dirty edge are identified. These test cases should be re-executed as the instruction sequences executed for the new and the old versions will differ.

In a practical tool the first phase is critical. Often commercial software binary programs are composed of tens or hundreds of software binary modules, each of which is composed of tens or hundreds of source code files. The relationship between software binary program and its source code files is often very complex. In order to be able to accurately identify modified software locations the exact source code of the software binary program is needed. We have overcome this problem by extending the compiler to instrument the code so that the intermediate representation control flow graph of the software binary module can be extracted. The final software binary program is composed from the instrumented software binary modules and thus it contains the instrumentation code, which ensures that the intermediate representation control flow graph of the

¹³ G. Rothermel, M. Harrold: A safe, efficient regression test selection technique. ACM transactions on software engineering and methodology, Vol. 6, No. 2, 1997, p. 173-210.

¹⁴ T. Ball: On the limit of control flow analysis for regression test selection. Proceedings of the 1998 ACM SIGSOFT international symposium on software testing and analysis, 1998, p 134-42.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 62 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

entire program can be extracted. As this information is part of the final software binary program, the symbols are resolved correctly. This is a prerequisite for correct source code level analysis. Note that this cannot be done before this complete software binary program has been link edited.

As to the second phase, the basic idea is that starting from the entry points of the new and the old version corresponding basic blocks are compared instruction by instruction. Say that basic block pair (b', b) has been reached through edges e' and e . If some instruction and its corresponding instruction are not equivalent, or if the outgoing edges of the based blocks do not match, then the edge e of the old version is said to be dirty, and the analysis does not continue through the outgoing edges of basic block pair. Otherwise, basic block pairs reached by corresponding edges are analysed.

The equivalence of instructions is not lexicographic but is based on type and usage of symbols occurring in the instruction. If those are equivalent, the instructions are equivalent if they are the same.

In the final phase the test cases that go through some dirty edge are identified in standard manner.

3.7.2 Innovation with respect to the STOA

The innovations of the work are related to techniques to build a practical tool. Firstly, extending compiler to instrument binary modules so that correctly resolved intermediate representation control flow graph of the whole binary program can be extracted is a key step. It is a prerequisite of accurate analysis but it also makes it easy to integrate the use of the method into existing software development processes.

Secondly, equivalence of instructions is usually based on lexicographic equivalence. In our case usage of symbols is used in determination of equivalence. This makes analysis more accurate, and independent of names of symbols. But it is also demanded by the use of intermediate representation control flow graph as it cannot be guaranteed that the compiler generated names are always the same.


3.7.3 Applications and contributions to case studies

The methods and tools were used in Metso case study. There were two different setups: a python based unit test environment, and robot framework based environment where complete process control station is tested. Technically from the test case selection perspective the environments are similar. Therefore we describe here only the unit test environment.

3.7.3.1 Python based unit test environment

This test environment uses python based unit test framework, and it is used to test function blocks, which are software binary modules of Metso's automation system offering.

Test cases are written as python classes. The function block under test is dynamically loaded into the python process, and then test case scripts are executed. The scripts use well-defined software interface of function blocks to execute various operations on the function block under test and to verify correct execution of those operations. The structure of setup is shown in Figure 26.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 63 of 80
		Version: 1.0
		Date : 23.05.2013
		Status : Final Confid : Public

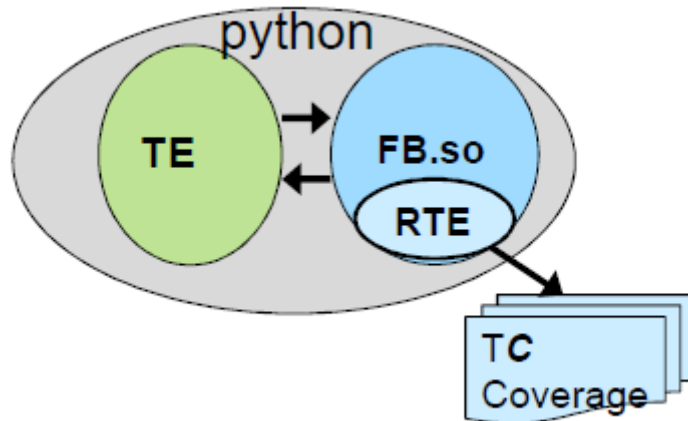


Figure 26: Python based unit testing environment

To create the instrumented sharable function block binary module two generic makefile-rules were added as shown in Figure 27. The first rule compiles a source code file with instrumentation activated and creates an instrumented binary module. The second rule creates the instrumented sharable function block module that can be dynamically loaded into python process (note that the rule uses `.oi` files that contains instrumented code).

```
%.oi: %.c
    SSA_BUILD_CFG=1 gcc $(CFLAGS) -fPIC -c $< -o $@

FB.so: $(patsubst %.c, %.oi, $(SRC))
    gcc -shared $? $(tc-sel-objs) -o $@
```

Figure 27: Makefile rules

A critical step in test case selection is identification of test cases. Our method requires that during execution of a test case the edges through which control flows are recorded. In practice this means that we can separate test cases from each other, i.e. we can say when a new test case starts. Usually this is the most challenging part as often test cases are executed one after another without initializing the software program under test. As a consequence when a fault is discovered, it is not clear which test case caused it. In this case separation of test case was done by adding an invocation of certain function of software interface of function blocks into the initialization function of the python test case class.

The control flow graph of the function block under test can be extracted with command

```
ssa-run -C TE.py
```

where `TE.py` is the test case executor which reads its configuration file from current directory. The program `ssa-run` is part of the test case selection system, and it instructs the run-time of the test case selection system to do what is needed. The control flow graph must be extracted separately for the new and the old version.


Edge coverage of test cases can recorded with command

```
ssa-run -E TE.py.
```

This must be done for the old version. Then the test cases going through a modified software location can be identified with command

```
Tc-sel -E ep_func -c FB1.cfg FB2.cfg FB*.ce
```

Where `ep_func` is the entry point (in this case it must be given separately as the system under test is just a library), `FB1.cfg` and `FB2.cfg` are the control flow graphs of the old and the new version of the function

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 64 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

block under test, and the files whose name start with `FB` and ends with `.ce` are the edge coverage files of test cases.

3.7.4 Metrics and results

3.7.4.1 Ease of integration

Based on the experience from the case study, integration of the source code change based test case selection tool into an existing development environment is easy. It can be done so that developers do not notice it at all. The most challenging part is separation of test cases. This is because test cases are usually executed one after another without initializing the system under test between test cases. In all three cases we have studied this far separation of test cases can be achieved by very small changes in test case definitions. In the python environment described above one line of code had to be added to initialization function of classes defining test cases. In the robot framework environment it was enough to add one line into the definition of a test suite. In the Ubuntu coreutils package we had to add few lines of code into the shell script running the test cases.

3.7.4.2 Effectiveness of selection

In the Metso case study VTT had test cases only for two function blocks whose software size was small, only few thousand lines of code or even less. We had several successive software versions of the function blocks. As the size of the function blocks were very small, the number of test cases going through a modified software location between two successive software versions varied a lot. A many cases zero test cases were selected, but in some case all test cases were selected. In average about 25% of test cases had to be re-executed.

As the software size of the function blocks we studied was small, we also studied feasibility of our method with Ubuntu coreutils package, which contains the basic commands of Linux operating system. Version 8.13-3.2ubuntu2.1 (which is the current version in Ubuntu release 12.10) contains a fix to avoid data-corrupting free-memory-read in certain situations. As described above, only minor changes in the test case executor script were needed in order to use our test case selection method.

The package contains 458 test cases. Only two of those are such that execution goes through a modified software location, i.e. only **5 per mil** of test cases should be re-executed. The test cases are scripts that invoke programs included into the package. The test cases execute 29427 times programs that are included in the package. Only five of those executions go through a modified software location, i.e. around **0.2 per mil** of program executions go through a modified software location.

3.7.5 Tool support

Currently only gcc (gnu compiler collection) version 4.5 is supported. Only programs written in the c-programming language are supported. It should be straightforward to add support for newer versions of gcc, and other programming languages supported by gcc.


3.7.6 Exploitation and dissemination

Active discussion on possibilities to start using the developed method and tools is going on with a company. Technical articles are in preparation.

3.8 FAULT DETECTION

3.8.1 Fault detection technique description

APIs often define abstractions of resources, such as files, and operations that can be used to modify these resources. Usually, such abstractions have a state and not all the operations are permitted at each state. This behaviour is seldom enforced by programming languages or the library itself, and faulty use of an API can result to undefined behaviour, subtle errors, or even crashes, which in the worst case may be utilized to

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 65 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

attack the system. Some of such errors may be hard to detect by traditional testing techniques, as they do not necessarily have visible effects on the expected output.

Our proof-of-concept fault detection tool verifies, at runtime, that an execution of a given C program uses certain programming abstractions according to a specified model. The expected use of APIs is modelled separately from the program itself and the models can be reused with other programs. The tool uses a compiler-based instrumentation to automatically track program's data flow and perform runtime checks about the state of instances of an abstraction before operations are permitted on them. The tool can be taken into use in existing development environments without changes to the software under development and only minimal changes to the environment. The tool can easily be run on top of existing test cases to detect faults. Furthermore, the tool facilitates debugging by producing dynamic slices of the program based on found faults.

3.8.2 Innovation with respect to the STOA

Similar API conformance checking techniques have been researched during the last decades. Related techniques include static and dynamic fault detection techniques. Static techniques, such as tpestate analysis and program rule mining, can consider all the possible executions, but often produce lot of false positives because of conservative approximation. Dynamic techniques are more precise, but often consider detection of faults related to only a single abstraction, such as locks. To our knowledge very few of these techniques have resulted as widely adopted tools.

Rather than extending the base technique, the research on the developed proof-of-concept solution considers the ease of use of the tool from the software engineering perspective. Issues considered include: easy and seamless integration into the development environment, extensibility of the tool (via new models), reusability of the models (separation from the program), ease of use by fully automated analysis, precision of the analysis (dynamic analysis, no false positives), and helpful error reports that facilitate debugging (dynamic slices).

3.8.3 Applications and contributions to case studies

Because of prolonged development of the tool, it has not contributed to any of the case studies. However, the tool has been successfully applied to several, widely used, operational software taken from the Ubuntu Linux distribution.

3.8.4 Metrics and results

A proof-of-concept tool was developed. It was demonstrated that the tool can easily be added into existing development system by only adding simple compiler wrappers, with no changes to the program or the build system.

3.8.5 Tool support

A proof-of-concept tool chain, consisting of instrumentation plug-in for GCC (prior art), a model compiler, and runtime libraries for dynamic fault detection, was developed during the project. In addition, a set of models describing C language file abstractions were designed.


3.8.6 Exploitation and dissemination

One publication describing the tool is in preparation by VTT provided in D5.WP5 deliverable.

3.9 OWASP-BASED WEB SECURITY TESTING

3.9.1 Technique description

Web applications are applications running on a web application server. The applications can be accessed using a browser from any computer which has access to the web application server. For the tests, the OWASP (Open Web Application Security Project) methodology is followed. OWASP is an open-source ap-

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 66 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

plication security project. It involves a community of corporations, educational organisations, and individuals from around the world creating freely-available articles, methodologies, documentation, tools, and technologies. The OWASP defined a testing guide that is a collection of all possible testing techniques along with an explanation of each one.

Web application tests are divided into 3 possible tests in the following order:

- Black box test

A so called black box test, where the tester knows nothing (or very little information) about the application, can be performed using the OWASP testing guide. Black box testing is a method of software testing that examines the functionality of an application (e.g. what the software does) without peering into its internal structures or workings.

This test is divided into 2 modes:

1. Passive mode: consists of trying to understand the application's logic, the network structure, the server name, IP, ports, HTTP requests, HTTP responses, system version (finger information of the system to test);
2. Active mode: consists of testing possible vulnerabilities using the OWASP testing guide categories and controls.

The black box test, also known as “vulnerability scan”, is more or less automated and is executed remotely. The test is executed using the OWASP testing guide which defines 9 categories with a total of 66 controls:

- Configuration Management Testing;
- Business Logic Testing;
- Authentication Testing;
- Authorisation Testing;
- Session Management Testing;
- Data Validation Testing;
- Denial of Service Testing;
- Web Services Testing;
- Ajax Testing.

Details on the OWASP methodology can be found in the OWASP testing guide.

- Grey box test

Grey box tests are performed like black box tests using the OWASP testing guide. This time however, the tester partially knows the internal structure of the application that is being tested. In some cases the tester is granted a valid account to the application.

- White box test

Procedure of a white box test


A white box test consists of a tester who has a detailed knowledge of the application and the internal network. He has access to all data he needs such as databases and source codes. The goal is to analyse and test a given application from inside to find possible vulnerabilities and to review the code to optimise and remove vulnerabilities.

- Manual application vulnerability tests

Web applications can be tested manually using the guide line of the OWASP testing guide and the expertise and experience of the testers. In this document we name the tests “custom vulnerability scans” even if each test is not a scan.

The custom vulnerability scans using the OWASP methodology pay particular attention to the top 10 application security risks. These risks have been identified as the most serious for a broad array of organisations:

1. Injections of any nature;
2. Cross site scripting (XSS);
3. Broken authentication and session management;
4. Insecure direct object references;
5. Cross site request forgery;
6. Security misconfiguration;
7. Insecure cryptographic storage;

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 67 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

8. Failure to restrict URL access;
9. Insufficient transport layer protection;
10. Invalidated redirects and forwards.

The custom vulnerability scans are not limited to any amount or type of tests and can be changed in order to provide tests which are customised to the customers' specific domain.

3.9.2 Innovation with respect to the STOA

There is no real innovation but the use of the testing technique based on OWASP combined with the expertise of the tester can lead to finding vulnerabilities which offer the opportunity to develop new innovative testing tools.

3.9.3 Applications and contributions to case studies

Custom tests based on the OWASP methodology were applied to the LASP case study.

3.9.4 Metrics and results

OWASP-based web security testing was used on the LASP case study conducted byitrust. The technique allowed us to provide the following recommendations:

- Put authentication on applications that don't need to be public;
- Fix or remove snort report application;
- Restrict access to the Tomcat manager.


3.9.5 Tool support

Tools used in the black, grey and white box tests:

Tool	Description	Category
nmap	A port scan sends customer requests to a range of server port addresses on a host, with the goal of finding an active port. Active ports can be exploited by attackers.	Port Scan
nslookup	This type of scan is used to determine vulnerabilities or locate malfunctioning devices on a network.	Network scan
dig	The domain information groper (dig), network administration command line tool for Domain Name System (DNS) queries and lookups.	
host		
traceroute	Network diagnostic tool that displays all network points that transmit the request from the computer where the command is launched to the destination.	
whois	A tool for checking information about ownership of a domain name using a domain or IP address.	
w3af	These tools are used to find and exploit web application vulnerabilities.	Web vulnerability scan
nikto		

Tools used in non-web application vulnerability scans:


Tool	Description
Nessus	A vulnerability scan assesses computers, computer systems, and applications for weaknesses that could be exploited by attackers. Areas that will be scanned: net-
OpenVAS	

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 68 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

Nexpose	work access, patch versions, system versions, application versions, system configurations, servers, server versions...
---------	--

3.9.6 Exploitation and dissemination

Custom tests based on the OWASP methodology have already been used for several ofitrust's customers.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 69 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

4. RISK ANALYSIS FOR RISK BASED TESTING

4.1 RISK-BASED TEST IDENTIFICATION AND PRIORITIZATION

We present a technique for risk-based test scenario identification and prioritization. The technique takes a risk graph as input, and produces a list of prioritized potential test scenarios as output. The prioritization can be automated. However, the risk graph may have to be annotated with information relevant for the prioritization in order to be accurate.

4.1.1 Technique description

In this section, we define the notion of a *risk graph* based on the notion of a *weighed graph*. We then define a function for prioritizing test scenarios based on a risk graph.

4.1.1.1 Weighted graphs

Before we define what is meant by a risk graph, we define a notion of a weighted graph. A weighted graph is a directed acyclic graph whose nodes and edges may be annotated by likelihood values. The nodes typically represent occurrences of events, and the likelihood value of a node specifies how likely it is that its associated event will occur. Edges represent causal relationships between nodes. Likelihood values of edges should be understood as *conditional likelihood values*.

In practice, there are many ways to specify likelihood values, e.g. as probabilities, frequencies, or intervals of these. Because of this, we parameterize the notion of a weighed graph by a notion of a *likelihood structure*. The advantage of this is that we do not have to specify separate rules for each possible way of specifying likelihoods.

Definition [Likelihood structure]: A likelihood structure LS is a tuple $(L, \oplus, \otimes, \mathbf{1}, \mathbf{0})$ consisting of

- A set L known as the likelihood values of LS ;
- Two elements $\mathbf{1}$ and $\mathbf{0}$ in L known as the maximum and minimum value of L , respectively.
- A binary operator \oplus on L , known as the or-operator or the sum-operator.
- A binary operator \otimes on L , known as the and-operator or the product-operator.

We denote by $LS.L$, $LS.\mathbf{0}$, $LS.\mathbf{1}$, $LS.\oplus$, and $LS.\otimes$, the likelihood values, the minimum value, the maximum value, the or-operator, or the and-operator of LS , respectively.

In practice, we will work with instances of likelihood structures. Therefore we have not imposed any constraints on the properties of L or on the operators.


To express likelihoods in terms of, say, probabilities, we have to instantiate the elements of the likelihood structure. Specifically, L will be defined the set of all real numbers between 0 and 1, $\mathbf{1}$ is defined by 1, $\mathbf{0}$ is defined by 0, \oplus is defined by + (addition), and \otimes is defined by * (multiplication).

Definition [Weighted graph] A weighted graph G over a likelihood structure LS is a tuple (Q, I, E, l) consisting of

- A set Q of states.
- A set of initial state $I \subseteq Q$
- A set of edges $E \subseteq Q \times Q$
- A function $l \in Q \cup E \rightarrow L$, assigning likelihood values to states and edges.

We denote by $G.Q$, $G.I$, $G.E$, and $G.l$, the states, initial states, edges, and likelihood assignment function of G . We sometimes just write Q, I, E , or l if G is clear from the context. We require that all weighted graphs be acyclic.

If $e = (p, q)$ is an edge, then the source and target states of the edge are defined by $src(e) = p$ and $tar(e) = q$. If G is a weighted graph and p is a state in G , then we denote by $src(G, p)$, all the edge in G that have p as target, i.e.,

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 70 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

$$src(G, p) := \{e \in G.E \mid tar(e) = p\}$$

To specify the test scenario prioritization function later, we need to be able to calculate the likelihood values of nodes in a graph based on the likelihood values of the edges. This is defined in the following.

Definition [Calculated likelihood of a state] Let G be a weighted graph over a likelihood structure LS where $LS.\oplus$ is commutative. Then the likelihood value of a state p in G , written $l[G, p]$, as calculated from the edges in G , is defined by

$$l[G, p] := \begin{cases} \bigoplus_{e \in src(G, p)} l(e) \otimes l[G, src(e)] & \text{if } src(G, p) \neq \emptyset \\ 1 & \text{if } src(G, p) = \emptyset \end{cases}$$

Note that the order in which the likelihood values are summed should not matter. Therefore we have required that \oplus must be commutative.

4.1.1.2 Risk graphs

In this section we define the notion of a *risk graph*, which is basically just a weighted graph whose nodes may be assigned consequence values in addition to likelihood values. We parameterize risk graphs with a notion of a *risk structure*.

Definition [Risk structure] A risk structure RS is a tuple (LS, C, rv) consisting of

- A likelihood structure LS ;
- A set of consequence values C ;
- A risk value function $rv \in LS.L \times C \rightarrow \mathbb{R}$ mapping likelihood values of LS and consequence values into risk values.

If RS is a risk structure, we denote by $RS.LS, RS.C, RS.rv$ the likelihood structure, the consequence values, and the risk value function of RS , respectively.

Definition [Risk graph] A risk graph G over a risk structure RS , is a tuple (Q, E, l, c) , consisting of

- A weighted graph (Q, E, l, l) over $RS.LS$;
- A partial function $c \in Q \rightarrow RS.C$ assigning consequence values of RS to states;


If G is a risk graph, then we denote by $G.Q_c$, or just Q_c if G is clear from the context, the set of all states in G that have a consequence value assigned to it. Furthermore, if e is an edge in G and l is a likelihood value of the likelihood structure of G , then the risk graph obtained by replacing the likelihood value of e by l is denoted by $we(G, e, l)$.

4.1.1.3 Test scenario prioritization

In this section, we describe a function for prioritizing test scenarios on a basis of risk graphs. We make the assumption that every edge in a risk graph is a potential test scenario. Intuitively, testing an edge e in a risk graph should be understood as testing the degree to which the system under test has any vulnerabilities that can be exploited in order to cause the event that is associated with the target node of e , given that the event associated with the source node of e has occurred.

We calculate the priority given test scenario (or edge) based on three values:

- **Severity:** An estimate of the impact that the test scenario has on the risks. As will be defined formally shortly, this value is calculated by comparing the risks of two risk graphs: one risk graph where the conditional likelihood of the relation corresponding to the test scenario is set to the minimum likelihood value (specifying that the relation will never lead up to something), and one risk graph where the conditional likelihood of the relation corresponding to the test scenario is set to the maximum likelihood value (specifying that the relation will always lead up to something). The severity is then the difference between the two risk graphs. A high severity value suggests that the test scenario has a high impact on the risks, and that it therefore should be prioritized for testing.
- **Testability:** An estimate of how testable a test scenario is. Typically, testability is an estimate of the effort required to implement the concrete test cases of the scenario given the tools and expertise available, but it can also be based on other considerations such as ethics. For instance, to test sce-

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 71 of 80
		Version: 1.0
		Date : 23.05.2013
		Status : Final Confid : Public

narios related to "Social engineering", one might have to "fool" employees of an organization, and this might not be considered ethical. The higher the testability, the higher the priority should be.

- **Uncertainty:** An estimate of how uncertain we are about the correctness of the conditional likelihood value of the relation corresponding to a test scenario. If the uncertainty is very low, then there might not be a need for testing, since then the test results may not give any new information. Conversely, a high uncertainty suggests that the test scenario should be prioritized for testing.

The severity value can be calculated automatically. However, values for uncertainty and testability must be annotated to the risk graph. To capture this, we define the notion of an annotated risk graph.

Definition [Annotated risk graph] An annotated risk graph G over risk structure RS , is a tuple (W, E, l, c, t, u) consisting of

- A risk graph (Q, E, l, c) over RS .
- A function $t \in E \rightarrow \mathbb{R}$ assigning testability values to edges.
- A function $u \in E \rightarrow \mathbb{R}$ assigning uncertainty values to edges.

The severity value of an edge e is obtained by calculating the difference in risk values between a risk graph G_{min} in which the likelihood of e is set to its minimum value and a risk graph G_{max} in which the likelihood of e is set to its maximum value. The intuition is that the higher this difference is, the more severe the edge is.

Definition [Severity] The severity of an edge e in a risk graph G , denoted $s(G, e)$, is obtained by calculating the impact that the edge has on the risk values of the risk models as follows

$$s(G, e) := \sum_{p \in Q_c} |rv(l[G_{min}, p], c(p)) - rv(l[G_{max}, p], c(p))|$$

where $G_{min} = we(G, e, 0)$ and $G_{max} = we(G, e, 1)$.

The priority of an edge is then defined by taking the weighted sum of its severity, testability, and uncertainty values.

Definition [Edge priority] The priority of an edge e in an annotated risk graph G is denoted by $p(G, e)$ and defined as follows

$$p(G, e) := (w_1 * s(G, e)) + (w_2 * G.t(e)) + (w_3 * G.u(e))$$

Where $w_1, w_2,$ and w_3 are weights that must be defined in advance.

Note that evaluating the appropriateness of using this function for calculating the priority based on severity, testability, and uncertainty is a topic of future work. Other functions, such as taking the product instead of the sum seem also to be reasonable.


Finally, the function that takes an annotated risk graph and produces a mapping that maps each edge (i.e. potential test scenario) to a priority is given by the following definition.

Definition [Prioritization function for annotated risk graph] The function p which takes an annotated risk graph G and produces a set of pairs of edges and priority values is defined by

$$p(G) := \{(e, p(G, e)) | e \in G.E\}$$

4.1.2 Innovation with respect to the STOA

Many approaches to risk-based testing have been proposed. Most of these approaches either (A) use risk assessment to prioritize areas in a systems that should be tested, or (B) use some kind of threat/fault modeling as part of the risk assessment and identify test scenarios on the basis of this. Approaches that follow (A) do not consider test identification (since only areas of a system and not tests) are prioritized, and approaches that follow (B) do not consider test prioritization. Furthermore, most approaches to risk-based testing that are described in the literature do not propose any new/novel technique that is specific to risk-based testing.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 72 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

Instead they rely on already existing techniques from risk assessment such as fault trees, misuse cases, HAZOP etc.

The only approaches that we are aware of that propose novel techniques that is specific to risk-based testing are the works of Chen et. al. [38][39], Stallbaum et.al [41], and Kloos et.al. [41]. Both Chen et. al. and Stallbaum et. al. rely on the assumption that a test model is already available (thus they do not consider test identification) at the start of the process, and they have two main steps: (1) annotate/update the test model based on risk information; (2) generate test cases from the test model and use risk information to prioritize tests. Kloos et. al. present a technique for generating tests based on a set of fault trees. This technique is the one that is the most similar to ours in the sense that it both considers test identification and test prioritization. One main difference between their technique and our technique is that theirs is based on fault trees, while ours is based on the more general notion of a risk graph.

4.1.3 Applications and contributions to case studies


The technique for test scenario identification and prioritization described in this section has been under development throughout the DIAMONDS project, and variations of it have been used in all the Norwegian case studies of the DIAMONDS project. In the first case study iteration, the technique had not been developed, and test scenario prioritization was performed informally. In the second iteration, a variation of the technique described here was used for manually prioritizing the test scenarios of a risk model. Because of the size of the risk model, this took about two days. In the third case study iteration, tool support was developed for automated prioritization. The two days of work was reduced to an hour. The technique used by the tool was a variation of the technique described in this section, and the insight gained by using the technique in the final iterator of the case study has led to the development of the current version of the technique.

In summary, our experience from the case studies suggests that having a technique for automatically prioritizing test scenarios is very useful since prioritizing the test scenarios manually can be very time consuming.

4.1.4 Metrics and results

Since the technique described in this section has been under continuous development during the project, we have not yet had the chance to evaluate the current version of the technique. However, we did make some observations when applying previous versions of the technique in case studies. One observation we made was that part of the risk model that described relevant test scenarios was somewhat small. In one case study iteration, about half of the risk model elements were not considered testable at all, and of those elements that were testable, about 1/3 were considered to have a very low testability. In addition to this, the testability of a test scenario tended to be either very high or very low. This had the consequence that the testability value had a large impact on the prioritization. One possible reason for this is that the level of abstraction of the risk model was quite high, and the risk model was not specifically developed to be used as a basis for test identification. Our aim was more to validate the parts of the risk model that could be tested.

There are in particular three aspects of the technique that we would like to evaluate as part of future work. First, we would like to evaluate the degree to which uncertainty w.r.t. the likelihood values in a risk model affects the test scenario prioritization. The likelihoods that we used in our case study were intervals of frequency values indicating the minimum and maximum frequency of occurrence. However, these kinds of likelihood values can result in imprecision when doing calculations (such as calculating the likelihood of a state based on the likelihood values of edges in a risk graph). Second, we would like to compare an earlier version of our technique with the current version documented in this section. The current technique is simpler and more intuitive. However, it would still be interesting to compare the results of the current technique with the previous version which was rather different from the current one. That version was based on propagating risk values from the leaf nodes in a risk graph to the edges in the diagram in order to find the severity value of an edge. Third, we would like to evaluate the how good our priority function for edges is. The current version is based on taking the weighted sum of the severity, testability, and uncertainty of an edge. However, we do not know how much sense this makes. For instance, it might be better to take the product of the severity, testability and uncertainty instead of the sum. This remains to be evaluated.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 73 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

4.1.5 Tool support

We have developed tool support for an earlier version of the technique described in this section. This was implemented as an extension of the CORAS risk assessment tool. We used the implementation as part of a case study in order to do automatic prioritization of test scenarios. However, further development is needed to make the implementation more user friendly and to adapt the current version of the technique.

4.1.6 Exploitation and dissemination

The CORAS approach to model-based risk assessment is a core part of SINTEF's expertise. The DIAMONDS project has enabled SINTEF to extend their work on risk assessment into the area of security testing, thus significantly strengthening the applicability and the validity of the approach. For SINTEF, this means improved opportunities for interesting future research as well as improved opportunities for consultancy related to risk assessment.

The technique described in this section relies on automated calculation of likelihood values in risk graphs. Such calculation is currently not supported in the CORAS tool for risk assessment, but will be implemented as part of the core CORAS tool as a result of the work in DIAMONDS. This will strengthen the applicability of the tool, both for SINTEF (who is developing the tool) and for other users of the tool. In addition to this, the technique for test scenario prioritization will be implemented as an extension of the CORAS tool.

4.2 RISK ASSESSMENT BASED ON VULNERABILITY ANALYSIS

4.2.1 Description

The complexity of modern day networks is overwhelming for people conducting security assessment in especially in the area of telecommunications. In an ideal world, all available interfaces would be tested, but in reality, budgets, deployment schedules and the availability of tools often impose limitations on what is feasible. In order to perform security testing efficiently yet thoroughly and reliably, it is necessary to analyse the network and prioritize the test targets. That way, the critical interfaces are properly tested and the resources are not wasted on testing issues that are trivial in the particular system under test. This paper will combine the benefits of recently published Unknown Vulnerability Management Lifecycle model, with other novel technologies such as Attack Vector and Attack Surface Analysis, and vulnerability metrics such as Common Vulnerability Scoring System (CVSS) to create a simple process for analysing the network, prioritizing the threats and planning the security testing accordingly.

4.2.1.1 Risk Assessment

Risk Assessment is a risk management phase where the risk of potential threat is determined. It is necessary to do the risk assessment before taking defensive actions in order to focus on issues that pose an actual risk. Most technical IT risks are based on the existence of vulnerabilities in the software, and the actual threat that the vulnerabilities pose. Therefore Oulu University Secure Programming Group (OUSPG) and Codenomicon have been using a simplified risk equation:

Risk = Threat * Vulnerability


The threat in the equation above refers to the likelihood that the vulnerability is exploited or otherwise triggered. However, not all realized threats are equally serious, and the consequences of an exploit vary. In practice, this equation is almost always complemented with "Value", or "Consequence" as seen for example in the Department of Homeland Security risk equation:

Risk = Threat * Vulnerability * Consequence

Risk assessment that builds on vulnerability analysis therefore attempts to measure the threats and the vulnerability of the system separately.

4.2.1.2 Attack vectors and Attack surface analysis

Organizations typically have complex networks with more exposed interfaces than they are aware of, and as a result they are be vulnerable to threats they do not know to exist. Comprehensive attack vector and attack surface analysis help in threat modelling.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 74 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

Threats are easiest to map using attack vectors. An attack vector is the means of exploiting a specific vulnerability in a system. Attack vectors include viruses, e-mail attachments, Web pages and so on. Attack surface analysis then connects the dots between attack vectors and real devices and software components, building a catalogue of products that need to be tested. The attack surface analysis is a starting point for mapping threats and understanding the exposure of a system.

4.2.1.3 CVSS vulnerability metrics

With the complex network structures and the heaps of vulnerabilities, it is necessary to have tools for evaluating and prioritizing the vulnerabilities.

The Common Vulnerability Scoring System (CVSS) provides an open framework for communicating the characteristics and impacts of IT vulnerabilities. CVSS consists of 3 groups: Base, Temporal and Environmental. Each group produces a numeric score ranging from 0 to 10, and a Vector, a compressed textual representation that reflects the values used to derive the score.

CVSS enables IT managers, vulnerability bulletin providers, security vendors, application vendors and researchers to all benefit by adopting this common language of scoring IT vulnerabilities. While the primary use of CVSS is to estimate the impact of realized vulnerabilities, CVSS scores provide an easy way to calculate numerical values for potential attack vectors.

4.2.2 Innovation with respect to the STOA

After interfaces are identified and analysed, vulnerability metrics such as CVSS will help prioritizing the interfaces for testing. Although CVSS was designed for completely different purpose, for evaluating the threat from reactive security findings by third parties, it also proves to be useful for proactive work where the actual vulnerabilities are still hiding, unknown to everyone.

In this study, the impact of compromising each of the affected interfaces was considered by applying the CVSS Impact Metrics and adding them to the Exploitability Metrics explained above. Impact Metrics are another component of the CVSS Base Score, and they provide a tool for estimating the impact of compromising a certain function or service. The CVSS Impact Metrics are the following:

- Confidentiality Impact (C)
 - None (N)
 - Partial (P)
 - Complete (C)
- Integrity Impact (I)
 - None (N)
 - Partial (P)
 - Complete (C)
- Availability Impact (A)
 - None (N)
 - Partial (P)
 - Complete (C)

The most severe attack would be one that could affect all three factors completely:

Confidentiality Impact: Complete

Integrity Impact: Complete

Availability Impact: Complete

Expressed in CVSS terms, the CVSS Vector for these terms would be:

C:C/I:C/A:C.

The overall CVSS Base Score is calculated by combining the Exploitability Metrics and Impact Metrics. An attack which might have score high in the Exploitability Metrics might only have a minor score in the Impact Metrics when considering the overall IMS deployment. For instance, attacking a WLAN Gateway might score high in the Exploitability Metrics, but it is questionable how serious the damage caused by compromising it could be to the complete IMS system.

The most severe attack would consist of the following factors:

Access Vector: Network
Access Complexity: Low
Authentication: None
Confidentiality Impact: Complete
Integrity Impact: Complete
Availability Impact: Complete

Expressed in CVSS terms, the CVSS Base Score for these factors would be 10, and the CVSS Vector would be:

AV:N/AC:L/Au:N/C:C/I:C/A:C.

4.2.2.1 Codenomicon Case Study: IMS

To test the risk assessment theory described above, it was used to analyse the IP Multimedia Subsystem (IMS) network interfaces. The attack vectors for complex telecommunications systems such as the IMS are difficult to identify and prioritize. This is due to the complexity of IMS architecture, which encompasses numerous different logical network entities and interconnections between them. This complexity of the architecture is the first challenge security experts face when preparing for a security audit in next generation telecommunication networks.

The IMS architecture is defined by logical network entities interconnected to each other by interfaces. In the study, based on attack surface analysis, certain IMS interfaces were selected for closer examination. They were then prioritized according to the Common Vulnerability Scoring System (CVSS) Exploitability Metric. The CVSS Base Score and the CVSS Vector format were used to create a prioritized table of IMS attack surfaces.

4.2.2.2 Codenomicon Case Study Results


The results show, that an attack which might have score high in the Exploitability Metrics might only have a minor score in the Impact Metrics when considering the overall IMS deployment. For instance, even though attacking a WLAN Gateway might score high in the Exploitability Metrics, it is questionable how serious the damage caused by compromising it could be to the complete IMS system, which lowers the WLAN Gateway overall CVSS Base Score.

Table 5 lists the selected interfaces with the related CVSS Vector and the Base Score.

Table 5: CVSS Vectors and CVSS Base Scores for IMS telecommunication core network.

Interface	CVSS Vector	CVSS Base Score
Gm	E- AV:N/AC:L/Au:N/C:P/I:P/A:C	9
Mb	F- AV:N/AC:L/Au:N/C:P/I:P/A:C	9
Mn	AV:N/AC:L/Au:S/C:P/I:P/A:C	8
Mi	G-AV:N/AC:L/Au:N/C:P/I:P/A:P	7,5
Ut	AV:N/AC:L/Au:S/C:N/I:P/A:C	7,5
Cx	AV:A/AC:M/Au:S/C:C/I:C/A:C	7,4
Mw	AV:A/AC:M/Au:S/C:C/I:C/A:C	7,4
Rf	AV:A/AC:M/Au:S/C:P/I:C/A:P	6,3
Mg	AV:A/AC:M/Au:S/C:P/I:P/A:P	4,9

The IMS case study not only showed how easy the prioritization is to do using CVSS, but also provide insight into the engineering and specification process as the people involved rarely consider such easy security metrics when building the architectures.

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 76 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

4.3 COMPOSITIONAL RISK ASSESSMENT

4.3.1 Technique description

Perfect security cannot be achieved in many applications in varying market sectors, including eCommerce, eGovernment, and eHealth. Trust allows people to use such applications though there will always be remaining risks. Before taking risks, it is reasonable to carefully analyse the chances, the potential benefits and the potential losses. Those offering security critical applications or services can use risk analyses to treat potential weaknesses in their products. Communicating the identified remaining risks honestly can be important to create trust. However, risk analyses might be difficult and expensive.

The technique presented here and described in detail in [46] introduces new concepts to reuse and combine results of the CORAS method [44][45] to make risk analysis more applicable for complex large scale systems. As an extension to CORAS it explicitly supports components by representing them with reusable threat interfaces.

Threat interfaces have vulnerabilities as input ports and unwanted incidents as output ports. Modelling the relations between the output and input ports of threat interfaces generated for individual components in a threat composition diagram, the probability values of unwanted incidents for the complex system composed of these components can be calculated. Gates representing mathematical functions (e.g. logical and, or) can be used to express how multiple unwanted incidents leading to the same vulnerability can affect that vulnerability so that other unwanted incidents are triggered. In the threat composition diagram, also external threats (e.g. human threats, natural disasters) and assets can be modelled and taken into consideration in a compositional fashion.

Conventional risk diagrams can be created directly from the threat composition diagram, which is the next step in the conventional CORAS risk analysis method. Nevertheless, it might be helpful to keep the component information for further analysis. The idea is to make components or complex combinations of components comparable in terms of risks. Typically, for a complex system, there is not only one single possible configuration. The system could probably be build using another combination of components or using completely other base components, too. It should be possible to choose the architecture with the fewest risks and to communicate such a design decision. Therefore, the component based risk comparison diagram is introduced in [46], which can be derived from the threat composition diagram.


4.3.2 Innovation with respect to the STOA

With gates and dependency sets the CORAS extension presented here can express relations that conventional CORAS diagrams cannot model well. This concept is similar to ETA and FTA [48]. But in contrast to a tree, there can be multiple top level incidents. Incidents can have relations leading to more than a single vulnerability having multiple consequence incidents. Therefore, dependencies can be modelled directly as common trigger nodes. In a fault tree, a fault triggering n other faults must be represented by n nodes having the same name but no graphical connection – which is less intuitive. Even more important, bouncing analysis becomes feasible by going top-down and bottom-up with the same model. Using FTA, bouncing analysis [42] is only possible in combination with other risk analysis methods like ETA or FMECA [43], which work on other models than fault trees. Transitions between different methods can cause problems and might be too difficult.

4.3.3 Applications and contributions to case studies

The technique evolved too late to be practically used within the DIAMONDS case studies and it is still under development.

Currently, it is applied for a large-scale trustworthy repository called the S-Network (<http://Surn.net>). The S-Network is going to provide guarantees for the long term preservation and for the permanent secure non-repudiation accessibility of its content. Requiring all users to agree on a user contract, the S-Network will offer legal validity for its content, including verifiable metadata values (e. g. who stored what and when) with standardized legal implications for all participants. The S-Network is intended to become a universal platform

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 77 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

for applications that have most stringent requirements, e.g. fair contract signing. Indeed, it must be resistant to both manipulation attempts and censorship. However, since it will not be possible to develop a perfectly secure solution, remaining risks have to be analysed and communicated in order to create trust in the S-Network. Due to its decentralized design of creating trust by distributing responsibilities over a set of mis-trust-parties [47], the S-Network will benefit from compositional risk analysis because it helps to identify threats affecting multiple parties if these share common components.

4.3.4 Metrics and results


For the S-Network, manipulative coalitions of humans belonging to different parties have been identified as a major threat using the risk analysis method shown here. Having effective treatments that prevent such collaboration of human threats is considered to be crucial. A more detailed game theory based analysis of the coalition threats with treatment measures for preventing these is presented in [47].

4.3.5 Tool support

Currently, there is no native tool support for the CORAS extension suggested here. However, tools for ETA/FTA can be used to do the calculation of probability values instead, but this requires manual creation of the trees. We expect that the CORAS tool for the conventional CORAS method (http://coras.sourceforge.net/coras_tool.html) will offer support for compositional CORAS in the future.


4.3.6 Exploitation and dissemination

The technique described here has been successfully presented at the PASSAT 2012 conference in Amsterdam. It is already used and developed further in other projects (e.g. RASEN, S-Network) at Fraunhofer FOKUS in Berlin.


	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 78 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

REFERENCES

- [1] P. Moore, R. J. Ellison, and R. Linger, "Attack Modeling for Information Security and Survivability," in Technical Note CMU/SEI-2001-TN-001, March 2001.
- [2] J. Williams and D. Wichers, "OWASP Top 10 2010," 2010, https://www.owasp.org/index.php/Top_10_2010-Main.
- [3] "OMG Unified Modeling Language Infrastructure Version 2.4.1," 2011, <http://www.omg.org/spec/UML/2.4.1/Infrastructure>.
- [4] "OMG Unified Modeling Language Superstructure Version 2.4.1," 2011, <http://www.omg.org/spec/UML/2.4.1/Superstructure>.
- [5] J. Bozic and F. Wotawa, "Model-based Testing - From Safety to Security," in *Proceedings of the 9th Workshop on Systems Testing and Validation (STV'12)*, October 2012.
- [6] J. Bozic and F. Wotawa, "XSS Pattern for Attack Modeling in Testing," in *Proceedings of the 2013 Eighth International Workshop on Automation of Software Test (AST'13)*, May 2013.
- [7] F. Wotawa, Trust But Verify!, Invited paper ASQT 2012 post-proceedings, to be published, 2013.
- [8] Cheng, S., Yang, J., Wang, J., Wang, J., Jiang, F.: Loongchecker: Practical summary-based semi-simulation to detect vulnerability in binary code. In: Proc. 10th Int. Conf. on Trust Security and Privacy in Computing and Communications, IEEE (2011) 150–159
- [9] Diamonds Consortium, section 2.2 "Vulnerability Directed Input Generation", Active Testing, deliverable D3.WP2, 2012.
- [10] A. Doupe, L. Cavedon, C. Kruegel, and G. Vigna, Enemy of the state: A stateaware black-box web vulnerability scanner," USENIX Security, 2012.
- [11] Hex-Rays, "Ida Pro disassembler and debugger", <http://www.hexrays.com/products/ida/index.shtml>
- [12] Khedker, U.P., Sanyal, A., Karkare, B.: Data Flow Analysis: Theory and Practice. CRC Press (2009)
- [13] P. Pietikainen, A. Helin, R. Puupera, A. Kettunen, J. Luomala, and J. Roning, Security testing of web browsers," 2011
- [14] Reps, T., Rosay, G.: Precise interprocedural chopping. In: Proceedings of the 3rd ACM symposium FSE. SIGSOFT '95, NY, USA, ACM (1995) 41–52.
- [15] Scholz, B., Zhang, C., Cifuentes, C.: User-input dependence analysis via graph reachability. In: IEEE Int. Workshop SCAM '08, Los Alamitos, CA, USA (2008) 25–34.
- [16] R. Sekar, An Efficient Blackbox Technique for Defeating Web Application Attacks," in *Network and Distributed System Security*, 2009.
- [17] G. Shu and D. Lee, Testing security properties of protocol implementations- a machine learning based approach," in *International Conference on Distributed Computing Systems*, 2007.
- [18] Z. Su and G. Wassermann, The essence of command injection attacks in web applications," in *Symposium on Principles of Programming Languages*, 2006, pp. 372-382.
- [19] Takanen, Ari, Jared DeMott, and Charles Miller (2008), "Fuzzing for software security testing and quality assurance", Artech House Publishers.
- [20] Zynamics, "Reil language specification," http://www.zynamics.com/binnavi/manual/html/reil_language.htm
- [21] Zynamics, "BinNavi - binary code reverse engineering tool", <http://www.zynamics.com/binnavi.html>
- [22] The Institute of Electrical and Electronic Engineers (1990) IEEE Standard Glossary of Software Engineering Terminology

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 79 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

- [23] OUSPG, (accessed on 4 December 2007) PROTOS Test-Suite:c07-sip, URL: <http://www.ee.oulu.fi/research/ouspg/protos/testing/c07/sip/#h-ref15>
- [24] Sutton M., Greene A. Amini P. (2007) Fuzzing Brute Force Vulnerability Discovery. Pearson Education, Inc. United States. 543 p.
- [25] Takanen A., DeMott J., Miller C. (2008) Fuzzing for Software Security and Quality Assurance. Artech House Inc., Norwood, MA. United States of America. 230 p.
- [26] F. Chen, G. Rosu, "Parametric trace slicing and monitoring," in: 15th Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2009, pp. 246--261.
- [27] C. Constant, T. Jeron, H. Marchand, V. Rusu, "Integrating formal verification and conformance testing for reactive systems," IEEE Trans. Software Eng. 33 (8) (2007) 558--574.
- [28] B. S. I. Group, "Bluetooth specification version 2.0 + edr [vol 0]," Tech. Rep., 1999.
- [29] D. Browning and G. Kessler, "Bluetooth hacking: A case study," in Proceedings of the Conference on Digital Forensics, Security and Law, 2009, pp. 20--22.
- [30] C. Gaston, P. L. Gall, N. Rapin, and A. Touil, "Symbolic execution techniques for test purpose definition," in 18th IFIP Testing of Communicating Systems (TestCom), 2006, pp. 1-- 18.
- [31] P. Mouttappa, S. Maag, A. Cavalli, "An iosts based passive testing approach for the validation of data-centric protocols," in: 12th International Conference on Quality Software (QSIC'12), 2012, pp. 49--58.
- [32] W. Reisig, Petri nets: an introduction. New York, NY, USA: Springer-Verlag New York, Inc., 1985.
- [33] Takanen A., DeMott J., Miller C. (2008) Fuzzing for Software Security and Quality Assurance. Artech House Inc., Norwood, MA. United States of America. 230 p.
- [34] A. Morais, and A. Cavalli, "A Distributed Intrusion Detection Scheme for Wireless Ad Hoc Networks", In Proceedings of 27th Annual ACM Symposium on Applied Computing (SAC'12), pp. 556-562, March 25-29, 2012, Riva del Garda, Italy.
- [35] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," Computer Networks, vol.31, no. 23--24, pp. 2435-2463, 1999.
- [36] A. Neumann, C. Aichele, M. Lindner, and S. Wunderlich, "Better approach to mobile ad-hoc networking (B.A.T.M.A.N.)", April 2008, IETF Internet-Draft (expired October 2008) [Online], available at <http://tools.ietf.org/html/draft-wunderlichopenmesh-manet-routing-00>.
- [37] A. Morais, and A. Cavalli, "An Event-Based Packet Dropping Detection Scheme for Wireless Mesh Networks", In Proceedings of 4th International Symposium on Cyberspace Safety and Security (CSS 2012), Melbourne, Australia, December 12-13, 2012
- [38] Y. Chen and R.L. Probert. A risk-based regression test selection strategy. In Proceeding of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE'03), Fast Abstract, pp. 305--306 (2003)
- [39] Y. Chen, R.L. Probert, and D.P. Sims. Specification-based regression test selection with risk analysis. In Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research, pages 1--14. IBM Press (2002)
- [40] J. Kloos, T. Hussain, and R. Eschbach. Risk-based testing of safety-critical embedded systems driven by fault tree analysis. In Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on, pages 26--33. IEEE (2011)
- [41] H. Stallbaum, A. Metzger, and K. Pohl. An automated technique for risk-based test case generation and prioritization. In Proceedings of the 3rd international workshop on Automation of software test, pages 67--70. ACM (2008)
- [42] Zigmund Bluvband, Rafi Polak, Pavel Grabov: Bouncing Failure Analysis (BFA): The Unified FTA-FMEA Methodology, ALD Tel-Aviv 2005

	Final Security-Testing Techniques Deliverable ID: D5.WP2	Page : 80 of 80
		Version: 1.0 Date : 23.05.2013
		Status : Final Confid : Public

- [43]Department of Defense: Proceedings for Performing a Failure Mode, Effects and Criticality Analysis, MIL-STD-1629, Washington 1949
- [44]Ida Hogganvik, Ketil Stølen: A Graphical Approach to Risk Identification, Motivated by Empirical Investigations, 9th International Conference on Model Driven Engineering Languages and Systems 2006, LNCS 4199 pp. 574-588, Springer 2006, DOI: 10.1007/11880240_40
- [45]Mass Soldal Lund, Bjørnar Solhaug, Ketil Stølen: Model-Driven Risk Analysis, The CORAS Approach, Springer 2011, ISBN: 978-3-642-12322-1
- [46]Johannes Viehmann 2012: Reusing Risk Analysis Results - An Extension for the CORAS Risk Analysis Method, 4th IEEE International Conference on Information Privacy, Security, Risk and Trust (PASSAT 2012), Amsterdam 2012, pp. 742-751, IEEE 2012
- [47]Johannes Viehmann 2012: The Theory of Creating Trust with a Set of Mistrust-Parties and its Exemplary Application for the S-Network, Proceedings of Tenth Annual Conference on Privacy, Security and Trust (PST), Paris (France) 2012, pp. 185-194, IEEE 2012
- [48]H. A. Watson: Launch Control Safety Study, Section VII, Vol 1, Bell Laboratories, Murray Hill 1961
- [49]Yixin Zhao, Jianping Wu, Xia Yin: From Active to Passive - Progress in Testing Internet Routing Protocols. J. Comput. Sci. Technol. (JCST) 17(3):264-283 (2002)
- [50]Alessandro Armando, Roberto Carbone, and Luca Compagna. LTL model checking for security protocols. Journal of Applied Non-Classical Logics, 19(4):403-429, 2009.
- [51]G. Rothermel, M. Harrold: A safe, efficient regression test selection technique. ACM transactions on software engineering and methodology, Vol. 6, No. 2, 1997, p. 173-210.
- [52]T. Ball: On the limit of control flow analysis for regression test selection. Proceedings of the 1998 ACM SIGSOFT international symposium on software testing and analysis, 1998, p 134-42.