Compositional Risk Assessment and Security Testing of Networked Systems

Deliverable D4.1.1

Baseline for Compositional Risk-Based Security Testing

RASEN - 316853

Project title:	RASEN
Project number:	316853
Call identifier:	FP7-ICT-2011-8
Objective:	ICT-8-1.4 Trustworthy ICT
Funding scheme:	STREP – Small or medium scale focused research project

Work package:	WP4
Deliverable number:	D4.1.1
Nature of deliverable:	Report
Dissemination level:	PU
Internal version number:	1.0
Contractual delivery date:	2013-01-31
Actual delivery date:	2013-01-31
Responsible partner:	Smartesting

Contributors

Editor(s)	Fabien Peureux (SMA)
Contributor(s)	Jürgen Großmann (FOKUS), Bruno Legeard (SMA), Fabien Peureux (SMA), Martin Schneider (FOKUS), Fredrik Seehusen (SINTEF)
Quality assuror(s)	Arthur Molnar (IW), Bjørnar Solhaug (SINTEF)

Version history

Version	Date	Description
0.1	12-11-09	TOC proposition
0.2	12-12-17	Preliminary draft of the deliverable
0.3	13-01-07	Updates of Section 2 & 3
0.4	13-01-11	Updates of Section 2 & 5
0.5	13-01-16	Updates of Section 4 & 5
0.6	13-01-21	Final updates for all document – ready for internal review
1.0	13-01-31	Final version

Abstract

Work package 4 will develop a framework for security testing guided by risk assessment and compositional analysis. This framework, starting from security test patterns and test generation models, aims to propose a compositional security testing approach able to deal with large scale networks systems. This report provides a state of the art of methodologies involved to reach this goal, respectively, risk-related security testing approaches, such as security testing metrics and testing approaches for large-scale networked systems. The report finally provides the RASEN baseline for compositional risk-based security testing. The baseline defines the basis for the development work to be completed during the project.

Keywords

Security testing, risk-based security testing, fuzzing on security models, security testing metrics, large-scale networked systems

Executive Summary

The overall objective of RASEN WP4 is to develop techniques for how to use risk assessment as guidance and basis for security testing, and to develop an approach that supports a systematic aggregation of security testing results. The objective includes the development of a tool-based integrated process for guiding security testing deployment by means of reasonable risk coverage and probability metrics.

The purpose of this document is twofold. First, we give an overview of the state of the art that is relevant for the WP4 research objective. Second, we present the RASEN WP4 baseline which is the existing tools and techniques that serve as a promising starting point for the research tasks.

The description of the state of the art and the baseline are organized according to the three main research tasks of WP4, which are the following:

- T4.1: Deriving test cases from risk assessment results, security test patterns and test generation models in a compositional way
- T4.2: Automating test execution based on risk assessment in a compositional way
- T4.3: Metrics and Dashboard of security testing results based on risk assessment

The discussion of the state of the art and the identification of the WP4 baseline are guided by the RASEN research questions that are relevant for the work package. These research questions are the following:

1. What are good methods and tools for aggregating test results (obtained by both active testing and passive testing) to the risk assessment?
2. How can test results be exploited to obtain a more correct risk picture?
3. What are good methods and tools for deriving, selecting, and prioritizing security test cases from risk assessment results?
4. What are suitable metrics for quantitative security assessment in complex environments?

As discussed in this deliverable, existing techniques do not deal with the complexity of large, heterogeneous networked systems. The progress beyond the state of the art targeted by RASEN focuses mainly on the compositional management of security testing, deriving security test results of the system from security testing of its components. This work will focus mainly on model-based security testing techniques, ensuring the compositionality on the basis of security test patterns and models.

Table of contents

1 Introduction

The objective of RASEN WP4 is to develop techniques for how to use risk assessment as guidance and basis for security testing, and to develop an approach that supports a systematic aggregation of security testing results. The objective includes the development of a tool-based integrated process for guiding security testing deployment by means of reasonable risk coverage and probability metrics. In reaching the objectives, WP4 focus in particular on three more specific tasks. First, developing techniques for deriving test cases from risk assessment results, security test patterns and test generation models in a compositional way. Second, developing tools for automating test execution based on risk assessment in a compositional way. Third, developing metrics and Dashboard of security testing results based on risk assessment.

This deliverable gives an overview of relevant state of the art, and identifies the baseline for the upcoming RASEN WP4 R&D activities. The state of the art in application security and vulnerability testing is structured in two main classes of techniques:

- **SAST** – Static Application Security Testing, which are white-box approaches that include source, byte and object code scanners and static analysis techniques;
- **DAST** – Dynamic Application Security Testing, which includes black-box web application scanners, fuzzing techniques and emerging model-based security testing approaches.

In practice these techniques are complementary, addressing different types of vulnerabilities. As we will see, DAST is most relevant for the RASEN project. The description of the state of the art in this deliverable and the identification of the WP4 baseline are guided by the relevant RASEN research questions. These, extracted from the RASEN DoW, include the following:

- What are good methods and tools for aggregating test results (obtained by both active testing and passive testing) to the risk assessment?
- How can test results be exploited to obtain a more correct risk picture?
- What are good methods and tools for deriving, selecting, and prioritizing security test cases from risk assessment results?
- What are suitable metrics for quantitative security assessment in complex environments?

The document is structured as follows. In Section 2 we give an overview of security testing approaches that are related to risk assessment. In Section 3 we present techniques to measure and quantify security testing relevance. Section 4 describes existing approaches to security testing of large-scale networked system. In Section 5 we present the RASEN WP4 baseline, before concluding in Section 6 by summarizing.

2 Risk-Related Security Testing Approaches

This section firstly presents the state of the art on risk-related approaches, focusing on Dynamic Application Security Testing (DAST). In Section 2.2, we motivate and describe techniques to identify and prioritize test cases with regards to risk analysis results. We next give an overview of the risk-based testing approaches that use security and vulnerability test patterns to drive the test case definition (Section 2.3), and that use security-annotated models to apply fuzzy techniques (Section 2.4).

2.1 Dynamic Application Security Testing

Software security testing aims at validating and verifying that a software system meets its security requirements [1][117]. Two principal approaches are used: functional security testing and security vulnerability testing [2][118]. Functional security testing is used to check the functionality, efficiency and availability of the designed security functionalities and/or security systems (e.g. firewalls, authentication and authorization subsystems, access control). Security vulnerability testing (or penetration testing, often called *pentesting*) directly addresses the identification and discovery of yet unknown system vulnerabilities that are introduced by security design flaws or by software defects, using simulation of attacks and other kinds of penetration attempts. **Vulnerability testing is therefore a risk-based testing approach** to discover vulnerabilities on the basis of exposed weaknesses and vulnerabilities in databases such as CVE[1], OWASP[2] or CAPEC[3] catalogues. These databases collect known vulnerabilities and provide the information for developers, testers and security experts. For example, the CVE database provides currently in NVD v2.2[4] more than 54 500 exposed vulnerabilities. They relate to all the technologies and frameworks used to develop web applications. For instance, more than 700 vulnerabilities have been identified in various versions of Joomla!, one of the Top 3 content management systems. But this large amount of vulnerabilities relates to a few number of vulnerability classes, such as cross-site scripting (XSS), SQL injection or file upload to mention some of the most prominent.

Model-based security testing. Model-based testing (MBT) uses selected algorithms for generating test cases automatically from models of the *system under test* (SUT) or of its environment. Although there are a number of research papers addressing model-based security (see e.g. [3][119][4][120]) and model-based testing (see e.g. [5]), there is still little work on model-based security testing (MBST). Of what exists in the state of the art, [6] discusses and implements an MBST approach which reads in context-free grammars for critical protocol interfaces and generates the tests by systematically walking through the protocol behavior. [7]The work in addresses the problem of generating test sequences from abstract system specifications in order to detect possible vulnerabilities in security-critical systems. The authors assume that the system specification, from which tests are generated, is formally defined using the Focus language. The approach has been applied to testing firewalls and transaction systems. In [8], a threat driven approach to model-based security testing is presented. UML sequence diagrams specify a threat model i.e., event sequences that should not occur during the system execution. The threat model is then used as a basis for code instrumentation. The instrumented code is executed using randomly generated test cases. If an execution trace matches a trace described by the threat model, security violations are reported.

Weakness and vulnerabilities models. A weakness or vulnerability model describes the weakness or vulnerability independently of the SUT. The information needed to develop such models is given by databases like the Common Vulnerabilities and Exposures (CVE) repository. These databases collect known vulnerabilities and provide the information to developers, testers and security experts, so that they can systematically check their products for known vulnerabilities. One of the challenges is how these vulnerabilities can be integrated into system models, so that they can be used for test generation. One possible solution is based on the idea of mutation testing [9]. For security testing, models of the SUT are mutated in a way that the mutants represent weaknesses or known vulnerabilities. These weakness or vulnerability models can then be used for test generation by

[1] CVE – Common Vulnerabilities and Exposures – MITRE - http://cve.mitre.org/
[2] OWASP – The Open Web Application Security Project - www.owasp.org
[3] CAPEC – Common Attack Pattern Enumeration and Classification – MITRE - http://capec.mitre.org/
[4] National Vulnerability Database Version 2.2 - http://nvd.nist.gov/ - Last access January 2013.

various MBT approaches. The generated tests are used to check whether the SUT is weak or vulnerable with respect to the weaknesses and vulnerabilities in the model. [10] presents the application of this principle to security protocols. [11] presents a methodology to exploit a model describing a Web application at the browser level to guide a penetration tester in finding attacks based on logical vulnerabilities (e.g. a missing check in a Role-Based Access Control (RBAC) system, non-sanitized data leading to XSS attacks). The authors provide mutation operators that reflect the potential presence of specific vulnerabilities and allow a model-checker to generate attack traces that exploit those vulnerabilities.

Fuzzing. Fuzz testing or *fuzzing* is a software testing technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions or memory leaks. Fuzzing is commonly used to test for security problems in software or computer systems. The field of fuzz testing originates with Barton Miller at the University of Wisconsin 1988 [12]. Fuzzing was originally based on a completely randomized approach. Recently more systematic approaches have been proposed. Black-box-based and model-based fuzzers use their knowledge about the message structure to systematically generate messages containing invalid data among valid inputs [13][129]. Systematic approaches are often more successful because the message structure is preserved and thus the likelihood increases that the generated message is accepted by the SUT. Fuzzing based on Security-Annotated Models, a relevant approach for the RASEN project, is presented in Section 2.4.

Penetration testing. Black-box web application vulnerability scanners are automated tools that probe web applications for security vulnerabilities, without access to source code. They mimic external attacks from hackers, provide cost-effective methods for detecting a range of important vulnerabilities, and may configure and test defenses such as web application firewalls [14][15]. Web application scanners have gained popularity due to their independence from the specific web application's technology, ease of use, and high level of automation. They have limitations, however, as discussed in Section 2.2.2.

Detecting errors – the oracle problem. Fuzzing and pentesting monitor for exceptions such as crashes, failing built-in code assertions or memory leaks. Test automation should provide better analysis of the test results, that is, automating the test oracle's evaluation of the test case's actual results as pass or no pass. One problem is the test oracle comparator: given a test case and an abstraction of expected results, how to automatically determine whether the Web application produces correct output. Major challenges to developing oracles for Web applications are the difficulty of accurately modeling Web applications and observing all their outputs [15][16].

2.2 Risk-Based Test Identification and Prioritization

This section deals with *techniques* that are used for identifying and prioritizing test cases based on risk analysis results. By technique we mean an algorithm, a language, or an analysis method which can be used as part of a methodology or an overall process. Most of the approaches to risk-based testing are based on already existing techniques within the areas of risk analysis (such as HAZOP or fault trees) and testing. The reader is referred to RASEN deliverable D3.1.1 for further details on these techniques. In the following we will concentrate on techniques that are specific to risk-based testing.

2.2.1 Motivation for Techniques for Risk-Based Test Identification and Prioritization

It is impossible to test every execution of most computer systems since there are usually an infinite number of possible executions. Therefore, when testing a system, we are usually forced to select the parts/executions of the system that will be tested. For security testing, where it is often difficult to find an adequate coverage criteria, the tester must choose which parts of the system to test based on some kind of prioritization of the most "security critical" parts of the system. Often this kind of prioritization is done informally by the security tester based on the knowledge that she/he has. The drawbacks of doing this informally are that:

- the decisions on which the prioritization is done are not well-documented;
- the prioritization will be extremely dependent on the security tester;

- the prioritization can be somewhat arbitrary and can be difficult to establish a sense of coverage.

In sum, these points could result in the testing effort being focused in the wrong directions, which again may result in the test cases being less likely to identify vulnerabilities in the system.

A structured approach to test prioritization is less likely to suffer from the same drawbacks as a purely informal approach. One natural way of structuring the prioritization process is to use the risk analysis to guide the test prioritization. This is natural because test prioritization can always be seen as being based on some notion of risk (although this notion may be implicit). In addition to providing a structured approach to prioritizing test cases, risk analysis can also be used as part of a test identification process. The reason for this is that a risk analysis is often performed by means of fault or threat modeling, and the results from this modeling can naturally be used as input to the test identification process.

2.2.2 Techniques for Risk-Based Test Identification and Prioritization

Although there are several approaches that use risk analysis in order to identify and prioritize tests, most of these approaches are based on already existing techniques from risk analysis and testing. In other words, very few new techniques are proposed that specifically combine risk analysis and testing. We will nevertheless give a summary of the existing techniques that are used by risk-based testing processes, before describing in more detail the techniques proposed that have been specifically developed in a risk-based testing setting.

Almost all the approaches to risk-based testing use risk analysis in one of two ways. Either the risk analysis is used to prioritize those parts/features of the system under test that are most risky, or risk analysis is used as part of a failure/threat identification process. The main steps of the former approach are:

- Step 1: Break the target of analysis into smaller parts/features.
- Step 2: Estimate a risk value for each part/feature.
- Step 3: Specify tests for the parts/features that have the highest risk.

Clearly, the approaches that follow this process use risk analysis to prioritize the testing. However, none of the approaches that follow this process use information from the risk analysis to identify relevant tests, only the test *areas* are identified. In Table 1 we have listed all the approaches that we are aware of that follow the process described above, and the techniques used in each step. All of these approaches use already existing techniques such as HAZOP [50] to identify risks, or code complexity measures to identify areas of code that are most likely to fail.

Approach	Step 1	Step 2	Step 3
Bach [17]	No particular support.	List of questions used in risk analysis. List and tables are used to organize risks.	No particular support.
Redmill [34][35][36]	System described as a set of services and service functions. No particular support for identifying these, but examples are given.	HAZOP is used to identify risks. Risk analysis based on likelihood and/or consequence with tables for documenting the results is proposed.	No particular support.
Souza et al. [39][40]	System described in terms of requirements and features	Risk analysis performed using questioners and/or checklists. Results are document using “metrics” which are proposed by the authors.	No particular support.
Bai et al. [18]	System described as a set of web-services.	Risks are calculated based on the probability of failure of service functions and so-called data ontologies which are arguments to service functions. Test cases are grouped according to the risk of the target features the tests are aimed at.	No particular support.
Felderer et al. [23]	System is described as a set of units, components, and features.	Criteria for how to measure the probability and impact are specified. Units, components, and requirements are estimated/measured according to the criteria. Finally, risks are calculated based on the estimated/measured values.	No particular support.
Ottevanger [32]	System is described as a set of quality characteristics and subsystems.	Select and determine relative importance of quality characteristics. Divide system into sub-systems and determine relative importance of these.	No particular support.
Rosenberg et al. [37]	Only object-oriented source code is considered, which is described/broken down into classes.	For each class, estimate the likelihood of failure based on a measure its complexity. Prioritize the classes with highest likelihood values when testing.	No particular support.
Wong et al. [43]	Only source code is a considered, which is broken down into functions or code blocks.	For each function of code block, estimate the likelihood of failure on a measure its complexity. Prioritize the functions/blocks with highest likelihood values when testing.	No particular support.

Table 1 – Summary of approaches that use risk analysis to prioritize test areas

The approaches that use risk analysis as part of a failure or threat identification have the following main steps:

- Step 1: Perform a risk analysis of the target system.
- Step 2: Perform threat/fault modeling.
- Step 3: Specify tests for the most severe threats/faults.

Many of the approaches that follow this process do not use the risk analysis in order to prioritize tests, but all of them use results from the fault/threat modeling as input to the test specification step. In Table 2, we have summarized all approaches that we are aware of that follow the above process, and highlighted the techniques used in each step. Again, most of the approaches are based on already existing techniques such as fault trees.

Approach	Step 1	Step 2	Step 3
Murthy et al [31]	Risk analysis is based on the NIST process [48].	Threat modeling activity is on Microsoft's Threat Modeling process from Microsoft's security developer center.	Misuse cases are used to describe security test scenarios. No detailed explanation of how to do this is given. No description of how threats are prioritized is given.
Zech et al. [44][45][46]	Performed by transforming the system model into a risk model. However, little description of system model, risk model, or the transformation is given.	Performed by transforming the risk model into a misuse case model. Little description of misuse case model or the transformation is given.	The misuse cases are seen as test cases that can be executed. Little description of how this is done is given. No description of how threats are prioritized is given.
Casado et.al [20]	Targets web-service transactions which are decomposed into properties. Fault trees [49] are used to describe risks (and threats) for each transaction property.		Leafs in fault trees correspond to tests. A test for a leaf node is a sequence of transaction notifications that will reach the state described by the leaf node. The details are a little unclear in the paper. No description of how tests are prioritized is given.
Kumar et al.[30]	Targets faults introduced by aspect oriented programming. A fault model is proposed (a table with fault types) and a risk model (a table assigning risk levels to fault types).		Tests are specified for fault types that have high risk. No description of how tests are prioritized is given.
Gleirscher [24]	The test model is derived from system requirements and expressed as a Golog script. Hazards/faults are specified as logic properties.		The test model is executed to see if it admits the specified hazards/faults. No description of how tests are prioritized is given.
Erdogan et al. [47]	Threats are modeled in CORAS [51] threat diagrams as part of the risk analysis.		Each threat scenario is prioritized based on three criteria: severity, testability and uncertainty. The threat scenarios with the highest priority are detailed into test cases.

Table 2 - Summary of risk-based testing approaches that use threat/fault modeling

The only approach which considers threat prioritization is [47]. In this approach, tests are derived from so-called threat scenarios that are prioritized according to three criteria:

- *Severity:* An estimate of the impact that a threat scenario has on the identified risks of the analysis.
- *Testability:* An estimate of the time it would take to test a threat scenario and/or whether the threat scenario could be tested given the tools available.

- *Uncertainty:* An estimate of the uncertainty related to the severity estimate of a threat scenario. High uncertainty suggests a need for testing

The only approaches that we are aware of that present novel techniques that are specifically intended to be used in a risk-based testing setting are [21][22][29][41]. All of these follow a process which differs slightly from the processes described above. That is, the approaches assume that a test model is already available at the start of the process, and they have two main steps:

- Step 1: Annotate/update the test model based on risk information.
- Step 2: Generate tests from the test model and use the risk information to prioritize the tests.

The approach given by Chen et al. [21][22] uses risk analysis for the purpose of prioritizing test cases in the context of regression testing. The part of the approach that considers the identification and prioritization of test cases is explained in detail and is motivated by its application on a case study.

The approach uses UML activity diagrams to model system features and then derives test cases from the activity diagrams. A test case, in this approach, is a path in an activity diagram starting from the activity diagram's initial node and ending at its final node. Furthermore, the approach carries out risk analysis of the test cases in order to prioritize them with respect to their risk exposure values. The risk analysis process in the approach is a slightly modified version of the risk analysis process given in Amland [1][16] and is therefore also supported by a table based risk analysis technique. The approach carries out this risk analysis process to also identify the most important test scenarios. Test scenarios are basically a collection of test cases that collectively simulate use cases. The approach does not specifically address security and is also is not supported by a dedicated tool.

The approach given by Stallbaum et al. [41] presents a technique to automate the generation of test case scenarios based on activity diagrams (which are regarded as test models in this approach), and to automatically prioritize test case scenarios based on their total risk value.

A test case scenario, in this approach, is a path in an activity diagram starting from the activity diagram's initial node and ending at its final node. This is similar to what is referred to as a test case by Chen et al. [21][22]. Furthermore, each activity in an activity diagram is annotated with a risk value. In this approach, a risk value is the product of the probability that an activity contains a fault and the total damage caused by the fault. The risk values that are assigned on the activities are further used to calculate the total risk value for a test case scenario. The approach does not explain how risk assessment is carried out, but focuses only on the automation technique.

Kloos et al. [29] present an approach for generating tests based on a set of fault trees (a fault tree represents one critical failure mode in its root node), and events that may lead to the critical failure mode in its child nodes and a so-called *base model*, i.e. a state machine providing information about the possible stimulations of the system under test. The main idea of the approach is to use the fault trees together with the base model to derive a test model which includes the event sets of the fault trees that are sufficient to test the root fault of the tree.

The following are the steps in the approach: (1) classify basic events of the fault tree, (2) select event sets of the fault trees, (3) build test model, (4) generate test cases and (5) execute test cases. Only Step 1, 2 and 3 are explained in detail.

The purpose of Step 1 is to match the events described in the fault tree with the abstraction level used in the base model by classifying all of the events in each fault tree into four different classes; "controllable", "observable", "external" and "internal" events.

In Step 2, the most critical failure modes are considered for further testing. It is mentioned that the most critical failure modes are identified based on the risk information given in their respective fault trees, i.e. fault probability and the severity given that the failure occurs, but it is not explained how, and based on what, these values are identified. All of the events in each fault tree are then grouped into event sets by making use of, what is referred to as, event set calculation. The purpose of Step 3 is to construct a test model. The test model is constructed by making use of a test model construction algorithm provided by the approach. The algorithm makes use of the event classification (Step 1) and transforms each event in the event sets (identified in Step 2) to state machine transitions. The transformed events are also integrated into the base model during this process. The final state machine diagram is then referred to as the test model.

In Step 4, test cases are generated based on the test model constructed in Step 3. The approach refers to other literature for details on generating the test cases from the test models, but they are basically generated by considering each path (starting from the initial state and ending at the final state) in the state machine diagram as a test case. Finally, the test cases are executed in Step 5.

2.3 Risk-Based Testing from Security and Vulnerability Test Patterns

While risk analysis is on one hand a system-specific task, on the other hand there are also vulnerabilities and threats common to a set of different systems or certain kinds of systems. E.g. common vulnerabilities for many systems are buffer overflows, or SQL injection for web applications. In the context of the ITEA-2 research project DIAMONDS[5], a set of templates for risk analysis was identified [68] based on security incidents [70] and security functional requirements from the Common Criteria [78]. Seven template diagrams were defined, e.g. “Malfunctions”, “External Intrusions and Attacks”, and “Malicious Behavior of Users and Administrators”.

The initial set of template diagrams were defined following a four-step process:

1. **Selection** of vulnerabilities and incidents from a list of 92 proposed security indicators.
2. **Mapping** of selected vulnerabilities and incidents to vulnerabilities and threat scenarios of the CORAS [88] risk analysis approach.
3. **Enriching** the template diagram with specific threats and vulnerabilities mentioned in the security indicator description selected in Step 1.
4. **Completing** the template diagram by manually mapping vulnerabilities and threat scenarios to security functional requirements from the Common Criteria. The security functional requirements act as treatments to the vulnerabilities and threat scenarios they are mapped to in Step 2.

An example of such a template diagram is depicted in Figure 1. It shows an excerpt of a “Malfunctions” diagram comprising several threats, e.g. “Hostile Administrator”, vulnerabilities, e.g. “Unlawful voluntary stoppage” and a threat scenario “INC21: Downtime or malfunction of the trace production function” based on a security indicator and two treatments, e.g. “FAU_GEN.1: Audit data generation”, from the Common Criteria.

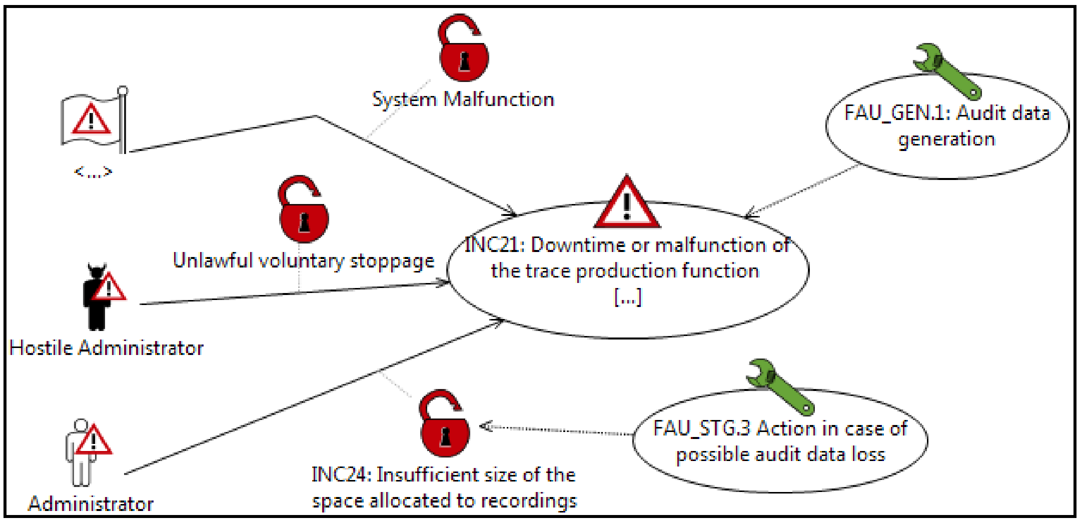

System Malfunction
FAU_GEN.1: Audit data generation
<...>
Unlawful voluntary stoppage
INC21: Downtime or malfunction of the trace production function [...]
Hostile Administrator
FAU_STG.3 Action in case of possible audit data loss
Administrator
INC24: Insufficient size of the space allocated to recordings

Figure 1: Example of a risk template [68]

[5] See www.itea2-diamonds.org for more information.

While these template diagrams are a way to facilitate model-based risk analysis, in addition to these risk templates a security test pattern catalogue was developed [67].

The SecurityTestPatterns.org group defines security test patterns as follows:

"A software security test pattern is a recurring security problem, and the description of the a test case that reveals that security problem, that is described such that the test case can be instantiated a million times over, without ever doing it the same way twice." [103]

The test pattern catalogue [67] comprises 17 security test patterns defined in a generic way and based on Mitre Corporation and the SecurityTestPatterns.org Group. Each security test pattern definition follows a tabular template for their description consisting of the following fields:

- Pattern name
- Context: kind of test pattern (e.g. organizational, generic, behavioral, or test data), group of security approach (planning, prevention, detection, diligence, response)
- Problem/goal: addressed testing problem or goal regarding security testing achieved by the security test pattern
- Solution: full description of the pattern
- Known uses
- Discussion: pitfalls, impact of the pattern
- Related patterns
- References

Table 3 shows the security test pattern “Verify Audited Event’s Presence” as an example. The fields “known uses” and “references” are here of interest. They may refer to external resources where the described pattern is applied or applicable. In this case, there are references to security functional requirements from the Common Criteria. In combination with the aforementioned approach of a template library for model-based risk analysis, security functional requirements forms a link between template diagrams and security test patterns and provide traces between risk analysis and security testing. This link allows applying security test patterns in order to conduct security testing based on risk analysis that takes advantage of the template diagrams.

Pattern name	Verify audited event's presence
Context	Test Pattern Kind: Behavioral Testing Approach(es): Detection
Problem/Goal	This pattern addresses how to check that a system logs a particular type of security-relevant event for auditing purpose
Solution	Test procedure template 1. Activate the system's logging functionality 2. Clear all existing log entries 3. Record current system time t_s 4. Stimulate the system to generate the expected event type 5. Check that the system's log contains entries for the expected event / Taking into account only logs displaying timestamps t_l satisfying the following condition: $t_l > t_s$
Known uses	Common Criteria SFRs: FAU_GEN.1, FAU_GEN.2
Discussion	This pattern assumes that the test framework provides means for tracing and evaluating the logs produced by the SUT. Evaluation may be performed online (i.e. quasi simultaneously, while the system is still running) or offline, i.e. after the system has completed its operation. An interesting issue to be considered is how to apply this pattern in situations whereby it may be impossible or too costly to clear the log repository or to restart the running system.
Related patterns (optional)	• Sandwich test architecture pattern • Proxy test architecture pattern • Verify audited event's content
References	FAU_GEN.1, FAU_GEN.2

Table 3: Example test pattern "verify audited event's presence"

2.4 Fuzzing Based on Security-Annotated Models

Fuzzing is a security testing approach consisting in stressing the interface of a system under test (SUT) with invalid, random or unexpected inputs [94]. The focus of fuzzing is to find security-relevant weaknesses in the implementation that may cause denial of service, degradation of service or undesired behavior [108].

Fuzzing experiences major development. The original fuzzers simply generated input data randomly. Such data is mostly totally invalid [108] and is therefore rejected by the SUT in most cases because it does not pass simple sanity checks [94]. To do so, modern fuzzers use knowledge about the protocol that is used for communicating with the SUT in order to generate input data. It aims at generating semi-valid input data, which means that it contains mostly valid parts, but also a few invalid parts, instead of being completely invalid. Hence, such data is called semi-valid [92]. Semi-valid input data has a significantly better chance to pass possibly faulty sanity checks. Containing only little invalid data, they may pass the sanity check of the SUT due to a single input validation fault, and therefore be processed by deeper components of the SUT. Semi-valid input is therefore often able to reveal security relevant weaknesses in the SUT's implementation [71][74][112][116].

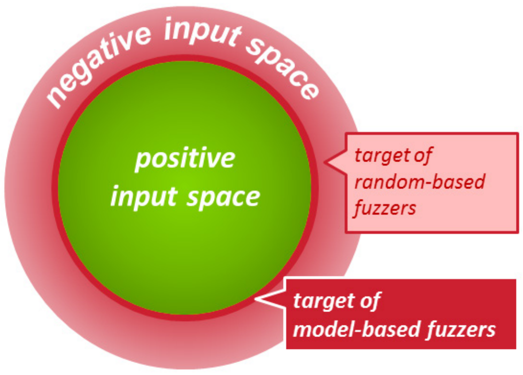


Figure 2: Targets of random-based and model-based fuzzers (cf. [108])

Model-based or smart fuzzers have a complete model of the SUT's protocol and are able to generate complex interactions with it by sending a set of valid messages and after reaching a certain state, stressing it with semi-valid input data. Therefore, while random fuzzers often find just simple bugs, more complex bugs can be found by model-based fuzzers [108]. Another consequence is that model-based fuzzers are more efficient because the targeted input space is much smaller than the input space of all invalid inputs targeted by random-based fuzzers. This is illustrated in Figure 2, which is adopted from [108].

While fuzzing is typically a black box approach where nearly nothing is known about the internals of the SUT [109], there are also some gray box and white box approaches where the source code or the code structure is used to guide the test generation process [71][73].

Determining the test verdict while fuzzing is much more difficult than by functional testing. While in functional testing the output of the SUT in reaction to a stimulus can be observed and compared to an expected result, this is not viable for fuzzing. More or less detailed monitoring of the SUT with respect to the expected faults is necessary. Simple monitoring can be achieved in different ways. One possibility is a connectivity check to verify if the SUT is still alive [105]. Another is valid case instrumentation where a functional test case is executed after each fuzz test case in order to determine if the SUT is not only alive but also working and functional [108]. Other kinds of bugs, e.g. memory corruption, require more sophisticated monitoring techniques, such as library interception that enables easier detection of memory bugs by using special libraries such as Guard Malloc [108].

2.4.1 Model-Based Data Fuzzing

As mentioned above model-based fuzzers employ protocol knowledge in order to generate semi-valid input data. The model can be created by a system engineer or a tester, or it can be inferred by investigating traces or using machine-learning algorithms. There are many possibilities for what can be used as a model. Context-free grammars are widely used as a model for protocol messages [53][84][112]. As a model for the flow of messages, state machines can be employed as in [53][57][59].

Grammars

Many of the early model-based fuzzers use context-free grammars for specification of both message syntax and the flow of messages. In the PROTOS project on Security Testing of Protocol Implementations [83][84], Kaksonen, Laakso and Takanen used a Backus-Naur-Form (BNF) based context-free grammar to describe the message exchange between a client and a server consisting of a request and a response, as well as the syntactical structure of the request and the response messages. The context-free grammar acts as a model of the protocol. In the first step, they replace some rules by explicit valid values. In a second step they insert exceptional elements into the rules of the grammar, e.g. extremely long or invalid field values. In the third step, they define test cases by specifying sequences of rules to generate test data.

In [112], Viide et al. introduce the idea of inferring a context-free grammar from training data that is used for generating fuzzed input data. They used compression algorithms to extract a context free grammar from the training data following the "Minimum Description Length" principle, in order to avoid the expensive task of creating a model of the SUT. The quality of the inferred model directly correlates with the amount and dissimilarity of available traces used for extracting the grammar.

While fuzzing is originally a black-box approach, Godefroid et al. [72] followed a white-box approach. In order to improve code coverage, they created a grammar from program branches by performing symbolic execution and constraint solving. For each program path, a path constraint is generated by combining all path conditions with logical “and”. These generated constraints are called grammar constraints. In combination with a context-free grammar of the input language, a context-free constraint solver solves these grammar constraints in order to generate inputs yielding to higher code coverage.

Kitagawa, Hanaoka and Kono [87] used a language called *tsfrule* for describing the state machine and message syntax. Messages are manually specified using regular expressions. They describe different fuzzing strategies they implemented: long string generation, violation of character restrictions, erroneous values, format string attacks as well as structural mutations affecting the number of fields or removing delimiters.

State Machines

In order to model the message flow of a protocol, also state machines can be used. Allen, Dou and Marin [55] modeled the FTP protocol in combination with block-based analysis. The state machine was used to generate so called test pattern that represents valid sequences of client/server interactions. In contrast, fuzzing of the different types of blocks is written manually in Python scripts.

Banks et al. describe in [57] a tool called SNOOZE for developing stateful network protocol fuzzers. The tool reads an XML-based protocol specification containing, among other things, the syntax of messages and a state machine representing the flow of messages. A fault injector component allows modifying integer and string fields to generate invalid messages. SNOOZE can be used to develop individual fuzzers and provides several primitives for fuzzing several values depending on their type. Monitoring of the system under test was realized by observing the received messages for their presence and the correct type.

Non-deterministic finite state machines (NFSM) are used by Jing et al. [82] for describing a protocol. The rationale for the use of non-deterministic finite state machines is that protocol specifications are often ambiguous. They focused on the verification of the SUT's state depending on whether a state transition is defined by a protocol specification for anomalous input. They rely on this differentiating factor and propose corresponding “normal-verification sequences”. These verification sequences are used to determine the SUT's current state and compare it to the protocol specification. Anomalous protocol data units (PDU) are generated by using e.g. boundary values, removing or adding fields within PDU or generating length and checksum errors. For each anomalous PDU, only one field was mutated at once called single-field mutation testing. Test cases are generated in TTCN-3 (Testing and Test Control Notation, version 3) with an extension. This extension consists of a new keyword *loopreplace* which allows modification of a template without generating a new one.

Shu, Hsu and Lee [77][104] used partial finite state machine (FSM) reduction to obtain a FSM from a large set of traces. They proposed to monitor the system to be tested to obtain traces and compute a tree FSM from these traces. In the next step, they minimized the number of states by calculating a set of merging constraints and use these to merge corresponding states. This learned FSM was used to guide the test process. Transition coverage was used to measure the comprehensiveness of the generated test cases. They implemented fuzzing itself by applying “typical fuzz function”.

For testing the IPv6 Neighbor Discovery Protocol, Becker et al. [59] used a finite state machine as a behavioral model of the protocol and decomposed the messages of the Neighbor Discovery Protocol. They applied several fuzzing strategies, e.g. changing field values or duplicating fields like checksums. They combined their fuzzing approach with reinforcement learning that used two reward functions: One function that measures the entropy, i.e. the number of different functions called while the SUT is processing a single message, and the power, i.e. the number of function calls due to processing a single message.

Taber et al. [107] used fuzzing to test softphones based on the Session Initiation protocol (SIP). For SIP messages, they used message templates that contain placeholders. These placeholders were replaced by well-known attack vectors and randomly generated data. A state machine contains the SIP states and guides a handler that generates the SIP messages. The softphone was controlled via its graphical user interface (GUI) by executing GUI events in the SUT. In contrast to other fuzzers, the GUI was also monitored for error messages or conspicuous changes.

Abdelnur, State and Festor [53] developed a fuzzer for the SIP protocol called KiF that uses an augmented BNF (ABNF) for the syntax of SIP messages and a protocol state machine for the transitions consisting of incoming and outgoing messages. A second state machine, called test state machine, was used to describe scenarios by prioritizing or omitting transitions. Syntactical valid and invalid messages are generated by employing a fuzzer expression grammar. This grammar is based on the ABNF and contains additionally a fuzzer evaluator. This evaluator allows the construction of syntactically invalid messages by replacing, repeating and choosing items. The SUT is observed by comparing received messages with the expected messages specified by the protocol state machine and checking the syntactical correctness using the context-free grammar. The aliveness of the SUT is detected by valid case instrumentation.

AutoFuzz is a fuzzer that acts as a proxy server. It was developed by Gorbunov and Rosenbloom [74] and constructs a FSM by observing incoming and outgoing messages with passive synthesis and partial FSM reduction. Additionally, the message syntax is captured in generic message sequences by employing algorithms that are used in bioinformatics for sequence alignment. Here, it was used to identify message fields for block-based analysis. AutoFuzz's role as a proxy server allows applying fuzzing functions to message fields depending on the state of the FSM.

Evolutionary Fuzzing

DeMott, Enbody and Punch followed this approach in [66]. They evaluate and evolve pools of sessions, where each session represents a complete transaction with the SUT, using a fitness function that determines the code coverage. The generations are created by crossing pools, selecting, crossing and mutating sessions. After creating a new generation, the SUT is fed with the sessions of the pools and the fitness of every session and pool is recalculated. This process is stopped after a given number of generations.

Duchene et al. [69] employed model inference assisted by evolutionary algorithms to evolve malicious input data in order to fuzz web applications for XSS vulnerabilities. They started by manually writing an attack grammar for cross-site scripting attacks. Individuals for the first generation are generated by this attack grammar and by reused input sequences from model inferring. The first generation evolves using a fitness function that assesses how close individuals are to detect XSS. How close individuals are to detect XSS vulnerabilities serves as a fitness function and is determined by the number of classes of the input grammar that are retrieved as output from the SUT and the traversed states. Successful XSS attacks are detected if the observed output is not confined by an output grammar.

A fuzzing approach that aims at risk-based security testing is presented by Xu et al. [115]. They understand risk as a relation between use case resources and targeted resources that are necessary to perform certain functionalities. The targeted resources are annotated in class diagrams. Sequence diagrams representing use cases are annotated based on the annotated class diagrams. After that, abstract test cases are generated by traversing the annotated sequence diagram. The process of test case generation is not described.

2.4.2 Behavioral Fuzzing Based on Security-Annotated Models

Behavioral fuzzing is a complementary approach to traditional data fuzzing. It aims at generating invalid message sequences instead of invalid input data in order to reveal vulnerabilities. An overview of behavioral fuzzing approaches, this section is originating from, can be found in [100].

Implicit Behavioral Fuzzing

Inferring a model from traces as performed in [112] leads often to implicit behavioral fuzzing. Depending on the quality of the traces, there may be substantial differences between the model and the implementation. Therefore, if the model is not exact, because the available traces have a poor quality, invalid message sequences may be generated and thus, implicit behavioral fuzzing is done when using the inferred model.

The evolutionary approach of DeMott, Enbody and Punch [66] in the previous section performs also implicitly behavioral fuzzing while learning the model. At the beginning of the learning process the model is mostly incorrect and so invalid messages and data are sent to the SUT. During the process, the learned model is getting closer to the implemented behavior of the SUT. During this approximation the fuzzing gets less random-based and gets subtler because the difference between the invalid generated behavior and the correct use of the SUT gets smaller. Therefore, implicit behavioral fuzzing performed by dynamic generation and evolution-based is superior to that performed by random-based fuzzers.

However, there is a crucial drawback of implicit behavioral fuzzing: While vulnerabilities like performance degradation and crashes can be found, other kinds of vulnerabilities cannot be detected. That is the case because there is no specification the revealed behavior of the SUT can be compared to and hence, vulnerabilities, e.g. revealing secret data or enabling code injection, may be perceived as intended features.

Explicit Behavioral Fuzzing

Behavioral fuzzing is mentioned in [84] where the application of mutations was not only constrained to the syntax of individual messages but also applied to “the order and the type of messages exchanged” [84]. Understanding behavioral fuzzing in that way, random-based fuzzing implicitly performs behavioral fuzzing. Because the protocol is unknown, randomly generated data can be both messages and data. Hence, in addition to data fuzzing, also behavioral fuzzing is done – but in a random way.

For testing the IPv6 Neighbor Discovery Protocol, Becker et al. in [59] used a finite state machine as a behavioral model of the protocol and decomposed the messages of the Neighbor Discovery Protocol. They applied several fuzzing strategies, e.g. changing field values or duplicating fields like checksums, which all constitute data fuzzing. The different fuzzing strategies mentioned by the authors are not constrained to fuzzing input data by deleting, inserting or modifying the values of fields. They are also supplemented by the strategies of inserting, repeating and dropping messages, which are already to be considered as behavioral fuzzing. Similar strategies are introduced in [77][104] where the type of individual messages is fuzzed as well as messages are reordered.

The aforementioned framework SNOOZE [57] provides several fuzzing primitives in order to develop a protocol-specific fuzzer. Among those primitives, there are functions to get valid messages depending on the state of a session and on the used protocol, but also primitives to get invalid messages. Thus, SNOOZE both data fuzzing and behavioral fuzzing.

The most explicit approach to behavioral fuzzing is found in [87]. Kitagawa, Hanaoka and Kono propose to change the order of messages in addition to invalidate the input data and find vulnerabilities. They employ a protocol specific state machine in order to determine whether a message shall be changed. Unfortunately, they do not describe how the message order should be changed to make it invalid.

An approach to model-based behavioral fuzzing using UML sequence diagram was presented in [100]. UML sequence diagrams, e.g. functional test cases, representing valid message sequences were modified by employing a set of behavioral fuzzing operators and applying them to model elements such as combined fragments, state invariants and time/duration constraints in order to generate an invalid sequence diagram. Fuzzing operators are for instance “Remove Message” that deletes a message from a sequence diagram, “Change Type of Message” that changes the operation that is called by a message or “Negate Interaction Constraint” that negates a guard of a combined fragment in order to activate the enclosed message sequence when it should not be activated. Table 4 shows a list of all fuzzing operators proposed in [100] and their constraints. Several of these fuzzing operators are being applied to a sequence diagram to generate a single test case.

Behavioral Fuzzing Operators for…	
Messages	Constraints
Remove Message Repeat Message Move Message Change Type of Message Insert Message	• not enclosed in a combined fragment *negative* • considered respectively ignored if enclosed in a combined fragment *consider/ignore*
Swap Messages	• not enclosed in a combined fragment *negative* • considered respectively ignored if enclosed in a combined fragment *consider/ignore*
Permute Messages Regarding a Single SUT Lifeline Rotate Messages Regarding a Single SUT Lifeline	• applicable for messages within a combined fragment *weak*
Permute Messages Regarding a Single SUT Lifeline Rotate Messages Regarding a Single SUT Lifeline	• applicable for messages within a combined fragment *strict*
Combined Fragments	Constraints
Negate Interaction Constraint Interchange Interaction Constraints	• applicable to combined fragments *option*, *break*, *negative* • applicable to combined fragment *alternatives*
Disintegrate Combined Fragment and Distribute Its Messages	• applicable to combined fragments with more than one interaction operand
Change Bounds of Loop	• applicable to combined fragment *loop* with at least one parameter
Insert Combined Fragment Remove Combined Fragment Repeat Combined Fragment Move Combined Fragment	• applicable to all combined fragments except *negative*
Change Interaction Operator	
Time/Duration Constraints	
Change Time/Duration Constraint	• applicable to constraints that are on the lifeline of the test component

Table 4: Behavioral fuzzing operators for UML sequence diagrams [100]

Because the approach generates a large number of test cases – usually too many to execute them all – a test case selection based on annotations to the model was introduced in [101]. The annotations were made using the UML profiling mechanism, in this case using the profile UMLsec [4] that provides a set of security related stereotypes. The UMLsec stereotype for specifying role-based access control was used and extended for UML sequence diagram and with additional tags for authentication. The achievement of this approach is twofold. On one hand, it allows focusing on a certain security aspect and thus allows to substantially reducing the number of test cases. In the presented example, the number of test cases was reduced from 48 test cases to 8. Another, much more important advantage is that this approach allows it to combine risk analysis and fuzzing by augmenting a model with stereotypes that refer to identified threats or vulnerabilities from the risk model.

3 Security Testing Metrics

How to measure security was reported to be a hard problem [80] and quantifying security in an absolute way is still considered an open question [92]. Savola et al. [99] state that the term security metric is misleading because security cannot be measured as a universal property. In spite of this, several definitions were made in the context of security, also by Savola [97]. (An exhaustive overview of different definitions of metric and measurement can be found in [58].)

The US National Institute of Standards and Technology (NIST) defines metrics as follows [106]: “Metrics are tools designed to facilitate decision making and improve performance and accountability through collection, analysis and reporting of relevant performance-related data. The purpose of measuring performance is to monitor the status of measured activities and facilitate improvement in those activities by applying corrective actions, based on observed measurements.” In a superseding document, they use the term measure instead of metrics for the “results of data collection, analysis, and reporting” and measurement for the process of measuring [64].

Vaughn et al. [110] argue that measures are often mistaken as metrics. They define metric as a relation of “individual measures to some common terms or framework”. Savola [97] stresses the difference between the terms "measurement result" and "metrics", and defines a measurement result as a “single-point-in-time data on a specific factor to be measured", while metrics are "descriptions of data derived from measurements”. In [80], Wayne Jansen summarizes several definitions of security metrics by saying that a “metric generally implies a system of measurement based on quantifiable measures”.

3.1 Measurements

All these understandings of metrics have in common the need for measurements. In order to get reasonable results from such measurements, they must fulfill some requirements:

- Reliability [86], repeatability [52], stability [61] or consistent measurement [81]: Measurements must be reliable meaning that different measurements using the same method in the same context have to be consistent. Reliability for qualitative measures is often difficult to achieve because certain security properties are intangible and subjective [56][80].
- Validity [86] means that the measurement should measure what it is intended to measure. According to Kan [86], validity consists of three properties: (a) construct validity means that actual measurements represent the theoretical construct, thus, it measures what is intended to be measured; (b) criterion-related/predictive validity describes how the values of measurements and the actual property to be measured relate and (c) content validity means the degree to which a measure covers the range of meanings included in a theoretical concept.
- Easiness [52] and cheap to gather [81], preferably in an automated way. If performing the measurement is too difficult compared to the information that can be obtained from it, there is no motivation to incur the measurement. Automated measurements reduce the opportunities for human error [64].
- Objectivity [52][110] and expression as a cardinal number or percentage [81], not with qualitative labels like ‘high’, ‘medium’, and ‘low’. Qualitative measurements are usually subjective [56] and hence, not appropriate. Obviously, subjective measures are interfering with the property of reliability. As discussed also by [95], in spite of this many security metrics are subjective.
- Use at least one unit of measure in order to be expressive and to declare what is measured [81].
- Contextually specific in order to allow decision making [52][81].
- Succinctness for reducing complexity, to remove aspects that are not important and thus, reducing the uncertainty of a measure [52].

Objective and, associated with it, quantitative security metric is doubted to be a feasible goal. A survey of quantitative security metrics by Verendel [111] revealed that most quantitative security metrics are not sufficiently validated by empirical tests. Hence, it is not clear whether they get corroborated or falsified.

Vaughn et al. [110] classified security metrics and measures along the following properties additional to objectivity as discussed above:

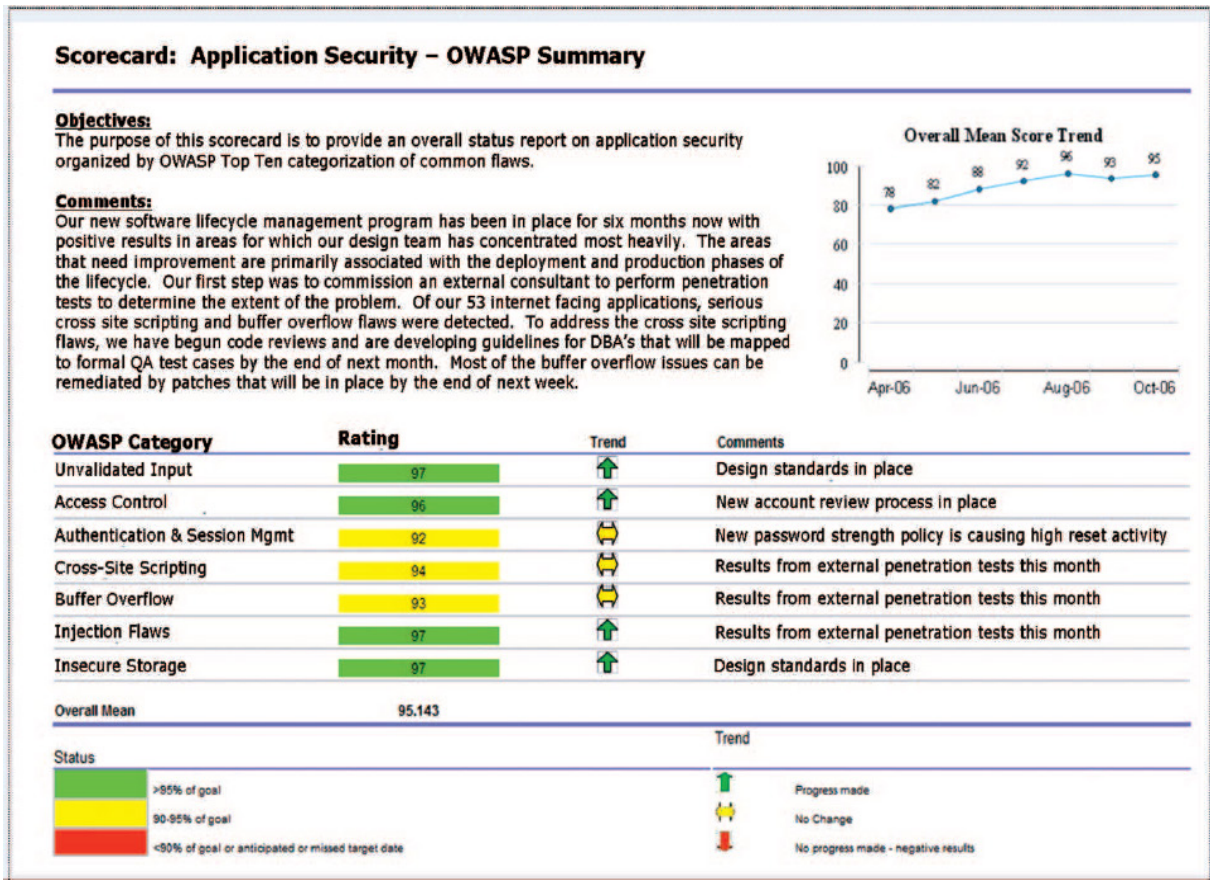
- Static/dynamic: Dynamic metrics evolve with time and are preferred over static metrics because the measured property of a system as well as the threats to a system [95] change over time.
- Absolute/relative: While relative metrics depend on other metrics, absolute metrics do not. Hence, relative metrics are meaningful only in the given context of the metrics they depend on.
- Direct/indirect: Direct metrics “are generated from observing the property that they measure”. This is the preferred way to obtain metrics, while indirect metrics can be used when direct measurement is not feasible.

3.2 Security Metrics Frameworks

As discussed above, there is an important difference between a measure and a metric. Vaughn et al. [110] describe the need for relating measures to common terms or a framework in order to obtain a metric. In the following, different approaches to achieve this goal are discussed. One is a security scorecard that describes in detail how measuring vulnerabilities can be composed to a security scorecard. Others are more abstract and describe general ways of composing and decomposing different metrics.

3.2.1 Security Scorecard

A simple framework was developed by Nichols and Peterson [91] in form of a security scorecard that summarizes different measurements related to the Open Web Application Security Project (OWASP) Top Ten Vulnerability Categories [96]. They propose to translate measurement results to a qualitative traffic light metaphor. For that purpose, they employed principles of the Six-Sigma-Quality Framework [79] for expressing values in terms of defects divided by opportunities, mapping them to colors and aggregating all different measures. The different measures, colors, and the aggregated measures including trends are depicted in an example in Figure 3, which is adopted from [91].


Scorecard: Application Security – OWASP Summary

Objectives:
The purpose of this scorecard is to provide an overall status report on application security organized by OWASP Top Ten categorization of common flaws.

Comments:
Our new software lifecycle management program has been in place for six months now with positive results in areas for which our design team has concentrated most heavily. The areas that need improvement are primarily associated with the deployment and production phases of the lifecycle. Our first step was to commission an external consultant to perform penetration tests to determine the extent of the problem. Of our 53 internet facing applications, serious cross site scripting and buffer overflow flaws were detected. To address the cross site scripting flaws, we have begun code reviews and are developing guidelines for DBA's that will be mapped to formal QA test cases by the end of next month. Most of the buffer overflow issues can be remediated by patches that will be in place by the end of next week.

OWASP Category	Rating	Trend	Comments
Unvalidated Input	🟩 97	⬆	Design standards in place
Access Control	🟩 96	⬆	New account review process in place
Authentication & Session Mgmt	🟨 92	⇔	New password strength policy is causing high reset activity
Cross-Site Scripting	🟨 94	⇔	Results from external penetration tests this month
Buffer Overflow	🟨 93	⇔	Results from external penetration tests this month
Injection Flaws	🟩 97	⬆	Results from external penetration tests this month
Insecure Storage	🟩 97	⬆	Design standards in place
Overall Mean	95.143		

Status		Trend	
🟩	>95% of goal	⬆	Progress made
🟨	90-95% of goal	⇔	No Change
🟥	<90% of goal or anticipated or missed target date	⬇	No progress made - negative results

Figure 3: Security scorecard [91]

3.2.2 Composition of Security Metrics

Composition of metrics is a controversially discussed topic. Though there is not much research, there is doubt if composition of security metrics is feasible. For instance, it is disputed in [80] that in theory systems can be designed to be predictably composable, while in practice this has never occurred; two systems can be composed, and in spite of the fact that both system are solely secure, the composition is not [80][110]. Verendel questions the simple addition of metrics for composed systems. He criticizes that mutual dependencies of components are neglected by additive composition of different security metrics. Savola [97] sees the necessity of decomposition of software security by design in order to evaluate the overall security of a system, but does not propose how this could help for measurement. Jansen [80] detected a lack of understanding and insight into the composition of security mechanism that impedes progress in security metrics. Composition approaches of security metrics is proposed either along the composition of a system, or along the composition of security requirements.

Composition of Security Metrics along Composition of a System

Wang and Wulf [114] propose a decomposition of a system (along physical subsystems or logical functions). They introduce three different functional relationships in order to determine the security strength of the composed element: *weakest link* if the functioning of the composed element is ultimately bounded by the weakest of its components, *weighted weakest link* as a generalization of weakest link with various degrees of impact on the different components to the composed element, and *prioritized siblings* where each of a set of components that constitute a composed element contributes to a different aspect of the composed element. For each of these relationships, Wang and Wulf provided a way how to calculate the security strength of the composed element from its components.

Walter and Trinitis [113] provide three different composition operators in order to compose metrics of different components depending on their interrelation: AND, OR and MEAN. AND is used when two components can be attacked in parallel where only one needs to be successfully attacked in order to compromise the system consisting of these two components – meaning that the composition is at most as secure as the components it is composed from. When two components compose two successive lines of defense and hence, both have to be attacked to compromise the composed system (meaning the composition is more secure than its components), the OR operator is used. The MEAN operator is used for systems where both components must be successfully attacked in order to compromise the composed system, but successfully attacking only one already compromises the system partially. Walter and Trinitis discussed the mathematical implications depending on statistical dependency and the kind of the (random) variable used to determine the security of any component. They also illustrate how more complex compositions can be achieved using these operators. However, the paper lacks an application that provides evidence of the approach as well as discussing the situations in where each of the operators are applicable. This remains unclear in particular for the MEAN operator.

Composition of Security Metrics along Security Requirements

Savola [95] argues for a taxonomy that enhances the composition of security metrics. He seizes the idea of Wang and Wulf [114] as discussed above, and applies it to security requirements as depicted by Figure 4, adopted from [97], in order to identify the basic measureable components. However, he does not focus on their interplay in order to determine the composition of metrics and measurements.

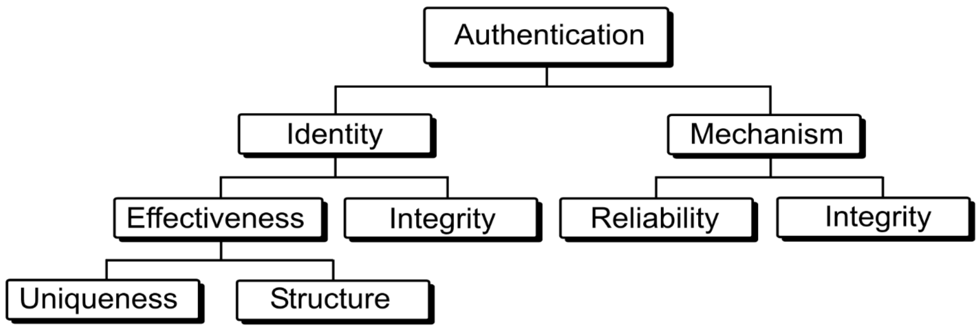


Figure 4: Decomposition of the security requirement "Authentication" [97]

A similar approach is used by Heymann et al. [75] to analyze the dependencies between security objectives that originally result from security requirements. They decompose security requirements and their corresponding security properties into three layers, one for high-level and low-level objectives, another for security patterns in order to achieve those objectives, and a third layer for the correspondent metrics. The benefit from this approach is twofold. On one hand, it explains how security patterns can be used to achieve certain security objectives and, therefore, the security requirements. On the other hand, it ships metrics with security patterns that facilitates measuring the corresponding security aspect. The decomposition of security requirements to security objectives is in turn used for composing the different metrics associated with the applied security patterns. The introduced decomposition operators are "and" and "or". Therefore, the decomposition is not as diversified as the decomposition approach by Walter and Trinitis discussed above.

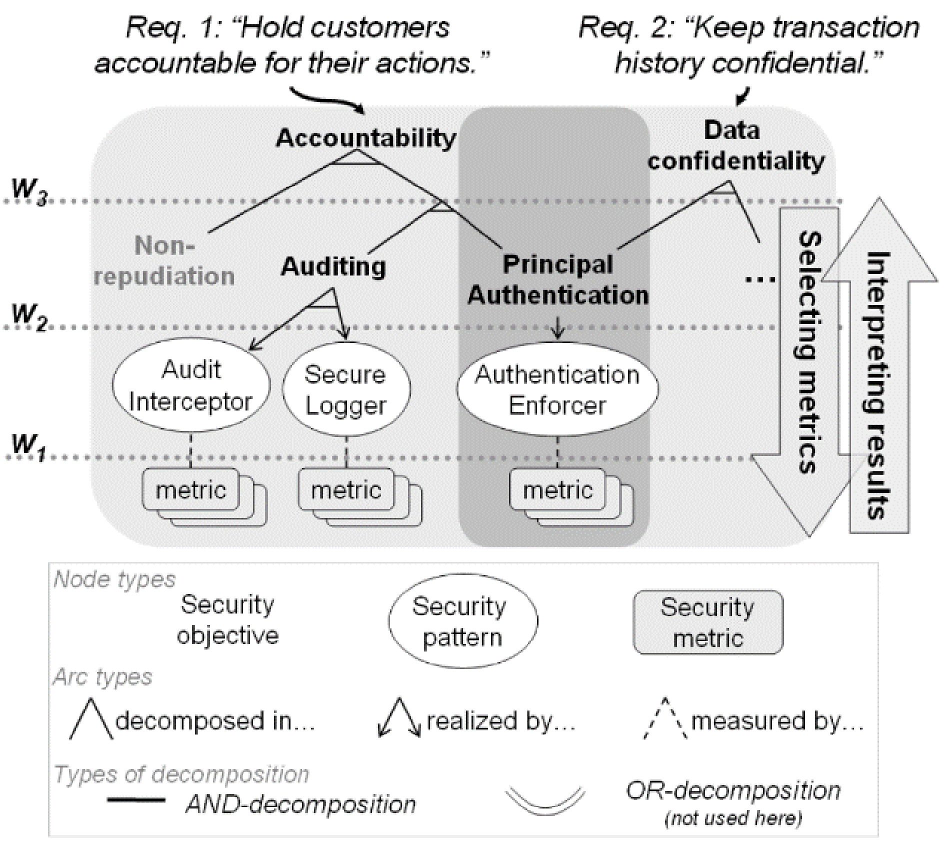


Figure 5: Composition of metrics along security requirements and patterns [75]

3.3 Metrics for Security Testing

As discussed in the previous sections, several required properties of security metrics have been identified, but Verendel [111] criticized the lack of validation of metrics. Jansen [80] noted that there are only few actual metrics that have been reported to be useful in practice. Many of such metrics refer to organizational measurements, e.g. [60] with metrics for baseline defense coverage, patch latency, password strength. Of course, some metrics are more general, whereas some are more specific for certain security testing approaches.

Many metrics regarding testing are coverage criteria. Some coverage criteria used for functional testing are also suitable for measuring non-functional security testing. Code coverage is a metric often used for testing and also for fuzzing. For example, Godefroid et al. [72] use instruction coverage, Abdelnur et al. [54] path coverage and DeMott et al. [66] unique function hits. The same approach is employed by Becker et al. [59] and by Abdelnur et al. [54] and is called entropy and power as measures of functions involved in processing a single message.

Another code coverage metric rather related to security is the attack surface [76][89][90]. It measures the attackability of a system by identifying its resources that constitute the system's attack surface. The approach is based on a formal entry point and exit point framework that identifies such points where (malign) data may enter or leave the system, and may act as basis of an attack [89]. The attack surface of a system is defined by these entry and exit points, the channels that can be used by an attacker to connect to the system, and untrusted data items the system reads from or writes to and the attacker has access to. An estimation of the damage potential of a resource based on its privileges and the effort the attacker has to spend to gain access to the resource is used to define the attack surface measurements. A weakness of this approach is that this damage potential-effort ratio is subjective and thus lacks objectivity as required by [52]. By measuring how test cases cover the attack surface of a system under test, a simple coverage criterion may be defined that can be used as a metric.

Nichols and Peterson [91] developed a metrics framework based on the top ten vulnerabilities of the Open Web Application Security Project [93]. In addition to design-time and deployment-time measurements, they propose runtime measurements for cross-site scripting, injection flaws and denial-of-service-vulnerabilities obtained by conducting penetration tests, e.g. fuzzing. This approach suffers from only respecting the most frequent vulnerabilities, just counting corresponding flaws and ignoring other aspects that may have a considerable effect on security.

There are also metrics specific for the security testing approach of fuzzing presented in Section 2.4. Microsoft made an empirical investigation based on Windows Vista file parsers for different file formats on how many test cases are necessary to cover most of the bugs. Microsoft recommends executing at least 500,000 test cases and another 250,000 test cases after each found bug. Using another fuzzing tool is recommended when no bug was found by the initial set of 500,000 test cases or after 250,000 test cases since the last found bug [102]. This poses a traditional rate of found defects that is also used in functional testing. But it seems questionable if these results are valid for other than the used tools and kinds of SUTs. Other metrics specific for fuzzing are coverage metrics regarding the specification. Kaksonen and Takanen [85] measure the ratio of symbols of a specification that were replaced by anomalies when generating invalid input data, and all symbols of the specification. They focus on locations of specification (the symbols) and disregard the kinds of anomalies (invalid values) injected on these locations. The different kinds of anomalies injected on the different locations can be measured related to the number of all anomalies applicable on the different locations [85].

3.4 Methodologies for Developing Security Metrics

While there are a few security metrics that are employable for security testing, as presented in the previous section, there are also methodologies for developing security metrics. Such methodologies may also be helpful for developing metrics for security testing.

One approach was developed by Chandra, Khan and Agrawal [62][63][64] and identified five phases:

1. **Identification of security factors** in order to find and extract a filtered set of security factors. Several activities relate to this phase, e.g. security requirements and validation, determining of security factor selection criteria, verification of security factors, and defining hierarchy and impact of security factors.
2. **Identification or design of a metric suite** comprising activities such as identification of software characteristics, analysis of available security models and metrics and designing security metrics and metrics suite.
3. **Validation of security metric suite** based on validation goals, standards, invalidity criteria, and validation framework. This phase also uses security factors identified in Phase 1 for validation.
4. **Quantification of security factors** as basis for the next phase.
5. **Estimation of security** is the last phase consisting among others of risk analysis, security ranking and precautionary measures.

The main weakness of the presented approach is insufficient description of how to conduct the the different phases. However, while they are intended for the estimation of security at design phase and not for security testing, they can serve as a template of a process of designing metrics for security testing.

Savola [98] proposes a risk-driven methodology for privacy metrics development consisting of the following steps:

1. **Privacy threat analysis** as analysis of risks for privacy.
2. **Utilization of privacy taxonomies** that help in creation of privacy objectives and requirements and may support development of metrics collections in a holistic and systematic way. The taxonomy to be used depends on the use case and on legal context.

3. **Development and prioritization of privacy requirements** in order to mitigate the threats identified in the first phase.
4. **Identification of basic measureable components** by decomposing the requirements identified in the previous phase. This approach was already discussed in Section 3.2.2.
5. **Development of measurement architecture** is technically necessary for obtaining and processing measureable data.
6. **Feasibility analysis and integration of metrics from other sources** on the basis of the identified basic measureable components.
7. **Balanced and detailed collection of privacy metrics** includes definition of
    a. their purpose,
    b. target description, e.g. using a composition-decomposition approach,
    c. formalization in a computable and understandable form,
    d. value scale or ordering,
    e. possibly needed thresholds.

This approach is focused on privacy along the example of cloud computing. But the risk-driven approach seems to be also appropriate for developing security testing metrics in the context of the RASEN project by adapting it to the needs for risk-based security testing. This may include:

- Step 1 and 3 because the CORAS approach used in RASEN also includes what is done in Step 3
- Step 2 for identification of suitable taxonomies,
- and the composition-decomposition approach used in steps 4 and 7 for target description of metrics.

4 Testing Approaches for Large-Scale Networked Systems

This section provides a state of the art of approaches that are used to produce security and/or vulnerability test cases for large-scale networked systems. Networked systems are today crucial components and often rely on a very high level of complexity since they support many necessary features from the basic functionality provided by computer networks to the applications executed on top of these networks. This complexity arises from a variety of causes. These systems are indeed often designed to address problems that cannot be completely defined: they meet the(rapidly changing) needs of diverse stakeholders; they must interact with other legacy systems, processes and policies; they may be critical systems that have to ensure both a high level of performance and dependability; and finally, they can be affected by political influences in the organizations developing the system or in the environment of the customer. Since networked systems have become complex and large-scale to be able to support increasing various needs and requirements, testing techniques targeting networked systems have to take this complexity into account. While there are many existing techniques for testing networked systems, each providing unique capabilities and benefits regarding vulnerability testing, few of them address large-scale features. We now provide an overview of existing techniques for testing networked systems, and introduce both their advantages and the limitations.

As proposed in [132], we can divide the different approaches into 3 categories: staging infrastructures, replay systems, and finally modeling and simulation-based approaches. In the following we look at each of them in turn.

4.1 Lab and Staging Infrastructures

Another technique for evaluating networked systems is deployment at a smaller scale before making the modified system visible to all users. Using this approach, the goal is to subject the system to an environment very similar that of the full production environment. Deploying applications at a smaller scale can be accomplished in multiple ways. The system can then be deployed to a staging environment that is intended to replicate or approximate the production environment. For example, Cisco maintains a testing facility called NSite [133] on which network devices and configurations can be tested before deployment. Infrastructures such as Emulab [134] provide a configurable environment for networked systems. Another methodology used in practice is deploying changes to a limited set of users. As instance, the large-scale distributed web infrastructure of Google uses this kind of approaches [135] in which a portion of users may be included in one or more concurrently-running experiments.

The primary benefit of lab and staging environments is that the actual networked system is executed without requiring modeling. In particular, it may be possible to capture bugs or test the system's performance under synthetic workloads or workloads captured from production systems. Another major benefit of staging environments using real users is that they can incorporate actual user behaviors, thus testing the full end-to-end workflow including the real networking environment.

This kind of approach also has important limitations. First, deploying and maintaining an alternate staging infrastructure may be costly. Such a staging infrastructure should be consistent with the production network in terms of not only configuration and topology, but also hardware, operating system, and software versions, since behavior may change between versions (e.g.[136]). Keeping such an environment synchronized with a production environment can be a large undertaking. Second, alternate staging environments may not fully capture the complexities of the production environment. Finally, allowing the tested system to serve real users may be risky if the system being tested does not perform as desired. Though it may be possible to reduce exposure by early detection of such cases, dissatisfied users may abandon use of a product or move to a competitor.

4.2 Logging/Replay Systems

Another useful technique for testing networked systems is logging and replay. Logging and replay consists of capturing data, typically an execution trace, from a running system in a production or staging environment, and then processing it offline. In an offline environment, it is possible to analyze the captured trace (e.g., for a visual display) or even replay the system to replicate the observed behavior and potentially ease the debugging process.

Replay tools typically strive to ease debugging for parallel and networked systems by providing deterministic replay. Networked systems are notoriously difficult to debug due to race conditions and non-deterministic behavior, making it difficult to detect and replicate bugs. Numerous approaches have been used to capture execution traces, spanning hardware-level (e.g.[137]), OS-level (e.g.[138]), user-space (e.g.[139]), and even cross-layer designs (e.g.[140]). The various approaches frequently make trade-offs in terms of logging overhead, types of applications they can support, whether they can replay a trace deterministically, and even relaxations on the replayed execution trace (e.g. [141]).

Logging and Replay have two primary benefits. First, they focus on using the actual system instead of a model. This allows developers to use the tools directly with existing systems with little or no modification. Indeed, some logging/replay tools are even targeted to be of low-enough overhead for production systems. Second, logging and replay tools typically provide repeatability. When debugging a complex networked system, the ability to replicate a bug and inspect the exact execution leading to an error can be extremely useful.

Despite their benefits, logging and replay systems do have limitations for testing large-scale networked systems. The primary limitation is that they can only consider scenarios that actually occurred; they are designed to capture execution traces and are not focused on testing alternative scenarios (e.g., by changing the behavior of the environment of the system itself). The ability to test alternative system behaviors or environments is important when evaluating a networked system (e.g., to evaluate behavior under particular failure scenarios).

4.3 Modeling and Simulation

Theoretical modeling and simulation techniques are characterized by capturing key properties of the full-scale system and creating an analytical model or an implementation focusing on key functionality. This has been a useful approach in many settings.

First, there are many generic simulation toolkits that have been developed and extended (see [142] for more details). These toolkits implement common functionality (e.g., network topologies and protocols) that approximate real-world settings. They frequently also provide interfaces to easily control and modify test scenarios. Using these toolkits, it is possible to develop a simulated version of a system and easily change parameters such as network topology, tests with machine failures, etc.

Theoretical modeling and analysis have also been widely applied in many networked systems. Due to the complexity of many networked systems, it is challenging to develop a tractable model for the complete system. Thus, many models are developed for specific parts of a system. For example, models have been developed for network reachability (e.g. [143]), security (e.g. [144]), interdomain routing (e.g., [145]), and content distribution in an application-layer overlay (e.g., [146]). There are only several works concerned with model based testing of large software. The bibliography analysis shows that the most part of model based testing society tends to work with toy and small-size examples (e.g. [147]), even if complexity and gigantic size of modern systems urgently requires new test development techniques. There only exists a limited number of works related with the large-scale model-based testing, but they mainly concern architecture-based testing (e.g. [148]).

By focusing only on key functionalities of a system, modeling and simulation have two primary benefits. First, since only key characteristics are captured, it can take less time to identify the impacts of certain changes. Making a change to the full-scale system typically entails details such as designing test cases, conducting regression tests, and handling error conditions. In addition, for networked systems in particular, it is typically faster to design, execute and analyze different test scenarios (e.g. network conditions, failure scenarios, etc.) since the physical infrastructure is not used. Second, modeling and simulation can be helpful to understand relationships between system components and parameters. Modeling and simulation also have important limitations. First, it can be difficult to determine exactly which key properties will affect the behavior of a complete system. The behavior of a networked system may not only depend on its own behavior, but also on the behavior of external systems that it uses (e.g., DNS, BGP). Furthermore, even if certain key properties are determined to be important, one may be forced to make simplifying assumptions to keep the model tractable or simply because it is not known how to model certain behaviors. For example, measurements studies (e.g. [149]) have shown that the behavior of certain networks can be very different than previously assumed. As a result, a model or simulation may miss possibly-unknown interactions with properties that are not modeled or not modeled accurately [150]. Beyond choosing which properties to model, it can also be difficult or impossible to keep a simulation or model in sync with a deployed system. As a networked system evolves, a simulation or model must be kept in sync in order to produce meaningful results.

4.4 Synthesis

Though existing techniques are useful, they appear to be not sufficient for realistically evaluating correctness of networked systems at a large scale. In particular, though lab and staging infrastructures may be able to run a networked system in various testing scenarios, they typically do not scale to the size and full complexity of the production environment. Logging and replay frameworks are limited to detecting bugs that have already been observed and are not suitable for testing alternative scenarios or system behaviors. Finally, abstract models used by simulation and modeling approaches do not capture the full complexity of many networked systems and the environments in which they run. To address these challenges within the RASEN project, we notably propose to use model-based approaches driven by risk assessment to focus the scope of the testing goal, and to adopt a compositional approach to manage the complexity of such systems. This focus would offer to increase the risk-relevance of the generated test cases and to propose scalable approaches for large-scale networked systems. The next section thus introduces the baseline to initiate the proposed RASEN approach.

5 RASEN Baseline

This section provides an initial assessment of the state of the art by specifying the starting point for the further development in the RASEN project.

5.1 Baseline for Techniques on Risk-Based Test Identification and Prioritization

In this section, we identify techniques for risk-based test identification and prioritization that will be used as a starting point for further research in the RASEN project. In the state of the art overview in Section 2 we distinguished between approaches that either (A) use the risk analysis to prioritize those parts/features of the system under test that are most risky, or (B) use risk analysis as part of a failure/threat identification process. All the approaches in category (A) address test prioritization, but not test identification. Conversely, all approaches in category (B) consider test identification and not test prioritization. However, there is one exception that considers both test identification and test prioritization [47]. The approach will therefore be carefully considered as a starting point for the R&D work of RASEN WP4. However, in [47], the technique for test prioritization is only described informally. Within the RASEN project, we aim to formalize the technique and to develop tool support for automated test prioritization.

As also discussed in Section 2, there are not many novel techniques that are proposed which are specifically intended to be used in the risk-based testing setting. All the approaches that we have particularly highlighted ([21][22][29][41]) are based on the assumption that a precise test model (in the form a UML activity diagram or state machine) is available at the start of the analysis process. This assumption restricts the applicability of the approaches. Therefore we will not initially consider any of these approaches as part of the RASEN baseline. It may, however, be the case that we later in the project consider a process where a precise test model is assumed. In that case, it would be appropriate to use the techniques proposed by Kloos et al. [29] as a starting point. Out of all the techniques proposed for risk-based testing, this one seems to be the most comprehensive.

5.2 Baseline for Risk-based Testing from Security Test Pattern

Security test patterns are a relatively new research area. The SecurityTestPatterns.org group maintains a catalogue of security test patterns with focus on the security problem that should be revealed by applying the pattern. Within the DIAMONDS project, an extended security test pattern approach focused on dynamic testing was presented [67]. The approach comprises nine security test patterns from the SecurityTestPatterns.org group and eight additional security test patterns. These security test patterns are enriched with known uses by security functional requirements from the Common Criteria.

On the other hand, a CORAS template library for risk analysis covering risks that are common for a large number of systems was developed [68]. Its purpose is to facilitate risk analysis using the CORAS approach. Beside common vulnerabilities and threats, also treatment measures in the form of security functional requirements from the Common Criteria.

The security functional requirements are the link between risk analysis and security testing when using templates from the CORAS template library. Thus, using risk templates that contain treatments based on security functional requirements, risk-based security testing is enabled.

While may serve as a starting point for risk-based security testing within Task 4.1—and thus, initially addresses the research question on good methods and tools for deriving, selecting, and prioritizing security test cases from risk assessment results—it provides no support for security testing based on risks that do not originate from any of the CORAS risk templates. Additionally, not all security test patterns reference security functional requirements from the Common Criteria. Further, not all security problems can be mitigated by functional security measures, such as errors in the implementation of functional security measures that may thwart security functional measures such as weak random numbers for cryptographic algorithms.

We thus propose to revisit and adapt the traditional approach of Model-Based Testing in order to generate vulnerability test from risk analysis to address large-scale system. This approach, called Risk-based Vulnerability Testing (RBVT), is depicted in Figure 6.

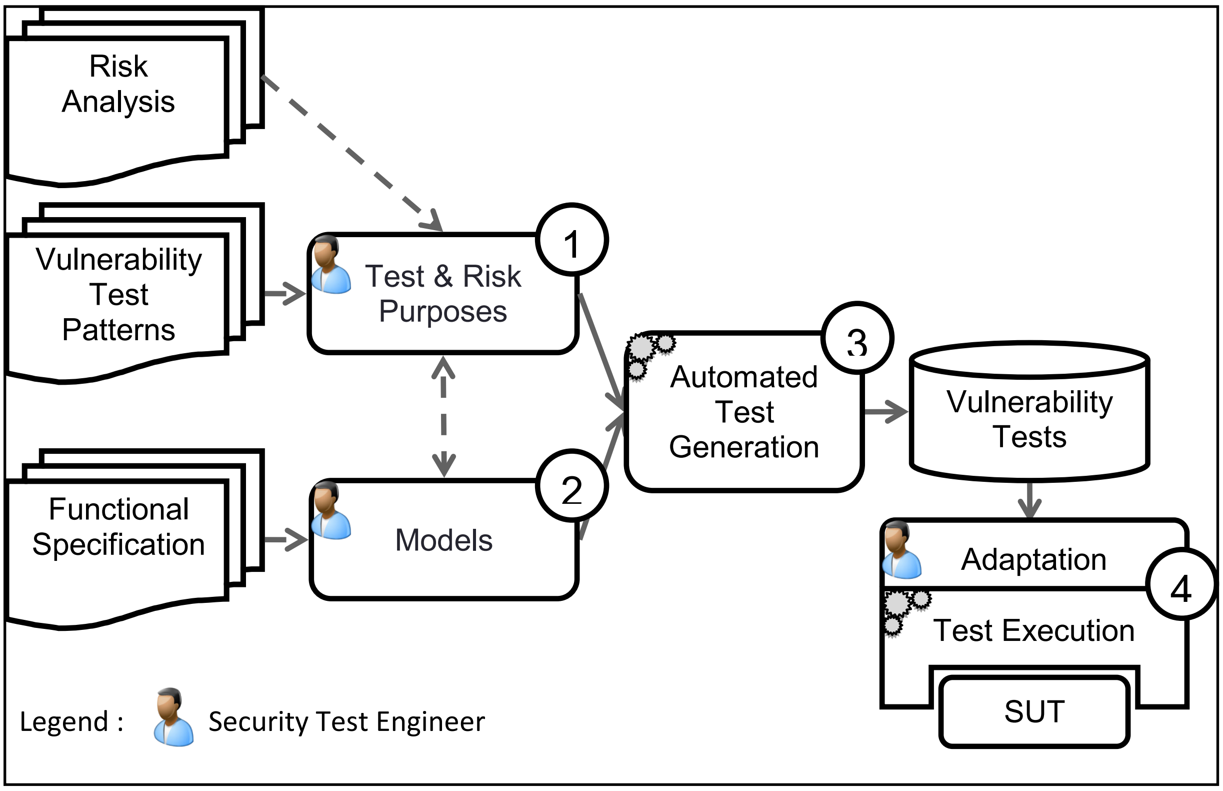


Figure 6: Risk-based vulnerability testing process

This process is composed of the four following activities:

1. Test and Risk Purposes activity consists in formalizing test purposes from risk analysis and vulnerability test patterns that the generated test cases have to cover.
2. Modeling activity aims to produce models that capture the behavioral aspects of the SUT in order to generate consistent (from a functional point of view) sequences of stimuli. The use of a composition of models should ease the management of large-scale system and enable the repeatability of the approach by compositionally reusing some results from sub-components when addressing a global system.
3. The Test Generation and Adaptation activity consists in automatically producing abstract test cases from the artifacts defined during the two previous activities;
4. The Concretization, Test Execution and Observation activity aims to (i) translate the generated abstract test cases into executable scripts, (ii) to execute these scripts on the concrete system, (iii) to observe its responses and to compare them to the expected results in order to assign the test verdict and automate the detection of vulnerabilities and risk reporting.

5.3 Baseline for Fuzzing Based on Security-Annotated Models

Fuzzing is an effective approach for finding vulnerabilities in systems. There are several approaches to guide the fuzz test generation for finding certain vulnerabilities, e.g. cross-site scripting vulnerabilities [69]. Despite that, there are no universal fuzzing approaches that are able to find different kinds of vulnerabilities and at the same time have guidance for how to identify specific kinds of vulnerabilities; fuzzing is still an approach to find different kinds of vulnerabilities without knowing in advance which will be found. Guiding the fuzz test generation process in order to find certain vulnerabilities or to focus on certain security aspects may be a way to reasonably reduce the number of fuzz test cases [101]

An approach to behavioral fuzzing [100] is presented that guides the test generation in order to find authentication bypass vulnerabilities [101]. For that purpose, a UML model is enriched with security annotations using the UMLsec profile with its stereotype for role-based access control. By extending the tags and applicability of this stereotype for UML sequence diagrams, guidance of the test generation approach is enabled. This allows testing for certain vulnerabilities and reduces the number of test cases and thus, improves the efficiency of the behavioral fuzzing approach regarding the tested vulnerability and enables risk-based fuzzing. In contrast to the risk-based testing approach discussed in Section 5.2, this approach relies on augmenting the behavior model with security-related information where testing for these vulnerabilities might be relevant. While the approach constitutes a starting point for the RASEN project, it has two drawbacks. On the one hand, behavioral fuzzing complements data fuzzing. It enables finding of new vulnerabilities that cannot be found with data fuzzing, and is a complementary approach to data fuzzing and should not be used solely. Therefore, the approach of risk-based testing using security annotations in the model has to be extended to data fuzzing. Additionally, there is only a single example for role-based access control. A systematic approach to the use of annotations for test generation guidance is an open issue that constitutes a starting point for a systematic risk-based fuzzing approach. A prioritization based on risk assessment results is also not addressed.

While there exists one approach to risk-based data fuzzing [115], its concept of risk as a relation between use case resources and target resources is different from the general understand of risk as the probability that a certain event may occur. In addition to this understanding of risk, the test generation process is not described.

5.4 Baseline for Security Testing Metrics

Task 4.3 in Work Package 4 of RASEN project is related to metrics for security testing results and addresses the RASEN research question on suitable metrics for quantitative security assessment in complex environment. Measuring security, and in particular quantitative assessment of security, is considered as a hard problem [80][92].

Section 3 presents state of the art of security metrics with respect to security testing, and also metrics frameworks to get an overall picture from a set of measurements and metrics, including composition of metrics and methodologies for developing security metrics. It can be generally said that there is only little research about metrics employable for security testing or composition of metrics regarding security. There exists only few security testing metrics where some of them are specific for fuzzing. Simplest are just counting the number of found flaws of certain kinds. Most are coverage metrics of code that does not respect the fact that, depending on the testing approach, the whole code cannot be covered. Hence, code coverage metrics becomes rather a relative than an absolute metric.

Another kind of metric more related to security is the attack surface metric [89] that uses the different resources of a systems (such as methods, channel and data) to determine the attack opportunities. Other metrics that are applied when conducting fuzz testing are related to the symbols of the specifications and the anomalies that are injected at the different locations of a specification identified by these symbols [85]. The combination of these two metrics could be a good starting point for the development of a coverage and completeness metrics that includes aspects regarding the results of the risk analysis that shall drive the security testing process. Additionally to security metrics, there are also a few methodologies for developing security metrics. One of these approaches is a risk-driven approach [99][98] (cf. Section 3.4) that seems to be a starting point for the development of such metrics as required by the RASEN project. For that purpose, that methodology has to be customized to the needs of the RASEN project.

Task 4.3 is furthermore about a dashboard for presenting the security testing results based on risk assessment. The security scorecard by Nichols and Peterson [91] as discussed in Section 3.2.1 goes in the direction of such a dashboard. While it focuses rather on found vulnerabilities than on security testing metrics in the sense of coverage and completeness as required by Task 4.3, the employed traffic light metaphor could be used for the dashboard for ease of use of use but requires definition of appropriate thresholds for the illustrated metrics.

6 Summary

RASEN WP4 addresses compositional security testing guided by risk assessment. This deliverable describes the state of the art of methodologies that provide guidance for security testing techniques from risk assessment. Section 2 provides an overview of risk-related security approaches, with a particular focus on risk-based testing techniques driven by vulnerability test patterns and security-annotated models. Section 3 provides an overview of security testing metrics and their composition along security requirements. The section also describes methodologies used to develop security metrics, which could be helpful for developing metrics for security testing within RASEN project. Section 4 gives an overview over existing security testing approaches that may be utilized to address large-scale networked systems. The section shows that, even if useful techniques exist, they mainly appear to be insufficient for realistically networked systems at a large scale.

The main lesson learned is that there is a lack of approaches of risk-based testing techniques (and associated tools) targeting security and vulnerability in the context of large-scale networked systems. However, all the related existing principles, mixed with the expertise of the project partners, permit to define a baseline for the RASEN methodology, as exposed in Section 5. For the further development in the RASEN project, this baseline proposes, as starting point, the following research topics in relation with the research questions enounced in the summary of the document:

- To use techniques addressing both risk-based test identification and test prioritization to drive the overall testing generation process. For this purpose, we propose to extend CORAS capabilities to allow the guidance of security test generation techniques on the basis of risk assessment results.
- To drive model-based testing generation using risk assessment results in order to generate vulnerability test purposes and test cases. This novel approach is called risk-based vulnerability testing (RBVT).
- To drive behavioral fuzzing testing approaches using models annotated by security information coming from risk assessment results. This novel approach is called risk-based fuzzing.
- To define a dedicated methodology for security testing metrics and to elaborate a dashboard for presenting the security testing results based on risk assessment.
- To propose a compositional security testing process enabling to deal with large scale networks systems in complex environments. This compositional approach leads to re-use results from sub-components when addressing a global module.

These baselines will be used within WP4 to elaborate a tool-based integrated compositional process for guiding security testing deployment by means of reasonable risk coverage and probability metrics.

References

[1] I. Schieferdecker, J.Grossmann, M. Schneider, "Model-Based Security Testing," in 7th Workshop on Model-Based Testing, vol. EPTCS 80 (2012)

[2] Shi Yin-sheng, Yuan F. Y. Gu, Tian-yang, "Research on Software Security," Testing World Academy of Science Engineering and Technology, vol. 69 (2010)

[3] Jürgen Doser, Torsten Lodderstedt, David Basin, "Model driven security: From UML models to access control infrastructures," ACM Trans. Soft. Eng. Methodol., vol. 15, pp. 39-91 (2006)

[4] J. Jürjens, Secure Systems Development with UML. Springer (2005)

[5] Zhen Ru Dai, Jens Grabowski, Ystein Haugen, Ina Schieferdecker, Clay Williams Paul Baker, Model-Driven Testing: Using the UML Testing Profile, 1st edition. Berlin: Springer (2007)

[6] R. Kaksonen, "A functional method for assessing protocol implementation security," VTT Technical Research Center of Finland, 448 (2001)

[7] J. Jürjens, Guido Wimmel, "Specification-Based Test Generation for Security-Critical Systems Using Mutations," in 4th Int. Conf. on Formal Engineering Methods (2002)

[8] E. Wong, Dianxiang Xu, Linzhang Wang, "A Threat Model Driven Approach for Security Testing," in Third Int. Workshop on Software Engineering for Secure Systems (2007)

[9] B. K. Aichernig, F. Wotawa, M. Weiglhofer, "Fault-Based Conformance Testing in Practice," Journal of Software and Informatics, vol. 3, pp. 375-411 (2009)

[10] P.-C. Heam, R. Kheddam, F. Dadeau, "Mutation Based Testing of Security Protocols in HLPSL," in Int. Conf. on Sofware Testing Verification and Validation (2011)

[11] J. Oudinet, A. Pretschner, M. Büchler, "Semi-Automatic Security Testing of Web Applications from a Secure Model," in 6th Int. Conf. on Software Security and Reliability, (2012)

[12] L. Fredriksen, B. So Barton, P. Miller, "An Empirical Study of the Reliability of UNIX Utilities," in Workshop of Parallel and Distributed Debugging (1990)

[13] J. DeMott, C. Miller, A. Takanen, "Fuzzing for software security testing and quality assurance," Artech House (2008)

[14] L. L. Pollock, H. Esquivel, B. Hazelwood, S. Ecott S. Sprenkle, "Automated Oracle Comparators for Testing Web Applications," in 18th IEEE Int. Symp. on Software Reliability, (2007)

[15] S. Amland. Risk based testing and metrics. In 5th International Conference EuroSTAR, volume 99, pages 8–12 (1999)

[16] S. Amland. Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study. Journal of Systems and Software, 53(3):287–295 (2000)

[17] J. Bach. Heuristic risk-based testing. Software Testing and Quality Engineering Magazine, 11:99 (1999)

[18] X. Bai and R.S. Kenett. Risk-based adaptive group testing of semantic web services. In Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International, volume 2, pages 485–490. IEEE (2009)

[19] A.F. Benet. A risk driven approach to testing medical device software. Advances in Systems Safety, pages 157–168 (2011)

[20] R. Casado, J. Tuya, and M. Younas. Testing long-lived web services transactions using a risk-based approach. In Quality Software (QSIC), 2010 10th International Conference on, pages 337–340. IEEE (2010)

[21] 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)

[22] 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)

[23] M. Felderer, C. Haisjackl, R. Breu, and J. Motz. Integrating manual and automatic risk assessment for risk-based testing. Software Quality. Process Automation in Software Development, pages 159–180 (2012)

[24] M. Gleirscher. Hazard-based selection of test cases. In Proceeding of the 6th international workshop on Automation of software test, pages 64–70. ACM (2011)

[25] International Standards Organization. ISO 31000:2009(E), Risk management Principles and guidelines (2009)

[26] International Standards Organization. ISO 29119 Software and system engineering - Software Testing-Part 2 : Test process (draft) (2012)

[27] D.W. Karolak. Software Engineering Risk Management (Practitioners). Wiley-IEEE Computer Society (1995)

[28] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. EBSE 2007-001, vol. 2 (2007)

[29] J. Kloos, T. Hussain, and R. Eschbach. Risk-based testing of safety-critical embedded systems driven by fault tree analysis. In Software Testing, Verication and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on, pages 26–33. IEEE (2011)

[30] N. Kumar, D. Sosale, S.N. Konuganti, and A. Rathi. Enabling the adoption of aspects-testing aspects: A risk model, fault model and patterns. In Proceedings of the 8th ACM international conference on Aspect-oriented software development, pages 197–206. ACM (2009)

[31] K.K. Murthy, K.R. Thakkar, and S. Laxminarayan. Leveraging Risk Based Testing in Enterprise Systems Security Validation. In 2009 First International Conference on Emerging Network Intelligence, pages 111–116. IEEE (2009)

[32] I. Ottevanger. A risk-based test strategy. IQUIP Informatica B. V, pages 1–13 (1999)

[33] Z. Qin, Y. An, and Y. Chen. Research of risk-based testing for web applications. Computer Engineering and Applications, 39(1):157–159 (2003)

[34] F. Redmill. Exploring risk-based testing and its implications. Software Testing, Verification and Reliability, 14(1):3–15 (2004)

[35] F. Redmill. Risk-based test planning during system development. In Proceedings of the 6th National Software Engineering Conference (2004)

[36] F. Redmill. Theory and practice of risk-based testing. Software Testing, Verification and Reliability, 15(1):3–20 (2005)

[37] L. Rosenberg, R. Stapko, and A. Gallo. Risk-based object oriented testing. In Proceedings of the 24 th annual Software Engineering Workshop, NASA, Software Engineering Laboratory. (1999)

[38] N.F. Schneidewind. Risk-driven software testing and reliability. International Journal of Reliability Quality and Safety Engineering, 14(2):99–132 (2007)

[39] E. Souza, C. Gusmao, K. Alves, J. Venancio, and R. Melo. Measurement and control for risk-based test cases and activities. In Test Workshop, 2009. LATW'09. 10th Latin American, pages 1–6. IEEE (2009)

[40] E. Souza, C. Gusm˜ao, and J. Venˆancio. Risk-based testing: A case study. In Information Technology: New Generations (ITNG), 2010 Seventh International Conference on, pages 1032–1037. IEEE (2010)

[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] E. van Veenendaal. Practical Risk-Based Testing{Product RISk MAnagement: the PRISMA method. Improve Quality Services BV (2009)

[43] W.E. Wong, Y. Qi, and K. Cooper. Source code-based software risk assessing. In Proceedings of the 2005 ACM symposium on Applied computing, pages 1485–1490. ACM (2005)

[44] P. Zech. Risk-based security testing in cloud computing environments. In Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on, pages 411–414. IEEE (2011)

[45] P. Zech, M. Felderer, and R. Breu. Towards risk–driven security testing of service centric systems.

[46] P. Zech, M. Felderer, and R. Breu. Towards a model based security testing approach of cloud computing environments. In Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference, pages 47–56. IEEE (2012)

[47] G. Erdogan, F. Seehusen, K. Stølen, J. Aagedal. Assessing the usefulness of testing for validating the correctness of security risk models based on an industrial case study. To appear in Proc. International Workshop on Quantitative Aspects in Security Assurance (QASA'12).

[48] RTI, Health, Social and Economics Research, The Economic Impacts of Inadequate Infrastructure for Software Testing, NC 27709 (2002)

[49] International Electrotechnical Commission: IEC 61025 Fault Tree Analysis (FTA) (1990)

[50] International Electrotechnical Commission: IEC 61882 Hazard and Operability studies (HAZOP studies) – Application guide (2001)

[51] M. S. Lund, B. Solhaug, and K. Stølen. Model Driven Risk Analysis - The CORAS Approach. Springer (2011)

[52] A. Atzeni, A. Lioy: Why to adopt a security metric? A brief survey. In: Quality of Protection. Security Measurements and Metrics. Advances in Information Security, vol. 23, pp. 1–12. Springer (2006)

[53] H. J. Abdelnur, R. State, O. Festor: KiF: a stateful SIP fuzzer. In: Proceedings of the 1st international conference on principles, systems and applications of IP telecommunications (IPTComm'07), pp. 47–56. ACM (2007)

[54] H. Abdelnur, R. State, J. Lucángeli, O. Festor: Spectral fuzzing: evaluation & feedback. INRIA, vol 7193 (2010)

[55] W. H. Allen, C. Dou, G. A. Marin: A model-based approach to the security testing of network protocol implementations. In: Proceedings of 31st IEEE Conference on Local Computer Networks, pp. 1008–1015. IEEE (2006)

[56] C. W. Axelrod: Accounting for value and uncertainty in security metrics. Information Systems Control Journal (6), (2008)

[57] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, G. Vigna: SNOOZE: toward a stateful network protocol fuzZEr. In: Proceedings of the 9th international conference on Information Security (ISC'06), pp. 343–358. Springer (2006)

[58] N. Bartol, B. Bates, K. M. Goertzel, T. Winograd: Measuring cyber security and information assurance, State-of-the-Art Report (SOAR). Information Assurance Technology Analysis Center (2009)

[59] S. Becker, H. Abdelnur, R. State, T. Engel: An autonomic testing framework for IPv6 configuration protocols. In: Proceedings of the Mechanisms for autonomous management of networks and services, and 4th international conference on Autonomous infrastructure, management and security (AIMS'10), pp. 65–76. Springer (2010)

[60] S. Berinato: A few good information security metrics (2005). [ONLINE] Available at: http://www.csoonline.com/article/220462/a-few-good-information-security-metrics [Accessed 10 December 2012]

[61] R. Böhme, F. C. Freiling: On metrics and measurements. In: Dependability Metrics. LNCS, vol. 4909, pp. 7–13. Springer (2008)

[62] S. Chandra, R. A. Khan: Software security metric identification framework (SSM). In: Proceedings of the International Conference on Advances in Computing, Communication and Control (ICAC3 '09).pp. 725–731, ACM (2009)

[63] S. Chandra, R.A. Khan, A. Agrawal: Security estimation framework: Design phase perspective. In: Information Technology: Sixth International Conference on New Generations (ITNG '09), pp. 254–259 (2009)

[64] E. Chew, M. Swanson, K. Stine, N. Bartol, A. Brown, W. Robinson: Performance measurement guide for information security. NIST Special Publication 800-55 Revision 1, National Institute of Standards and Technology (2008)

[65] H. Dai, C. Murphy, G. Kaiser: Configuration fuzzing for software vulnerability detection. International Conference on Availability, Reliability, and Security (ARES'10), pp. 525–530 (2010)

[66] J. D. DeMott, R. J. Enbody, W. F. Punch: Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. Black Hat USA and DefCon 2007 [ONLINE] Available at: https://www.blackhat.com/presentations/bh-usa-07/DeMott_Enbody_and_Punch/Whitepaper/bh-usa-07-demott_enbody_and_punch-WP.pdf [Accessed 11 December 2012]

[67] DIAMONDS: Initial security test patterns catalogue. DIAMONDS project deliverable D3.WP4.T1

[68] DIAMONDS: Initial methodologies for model-based security testing and risk-based security testing. DIAMONDS project deliverable D3.WP4.T2-T3

[69] F. Duchene, R. Groz, S. Rawat, J. Richier: XSS vulnerability detection using model inference assisted evolutionary fuzzing. In: IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST'12), pp.815–817. IEEE (2012)

[70] European Telecommunications Standards Institute (ETSI): Information Security Indicators (ISI), Part 1.1: A full set of operational indicators for organizations to use to benchmark their security posture, Version 0.1.5, November 2012

[71] V. Ganesh, T. Leek, M. Rinard: Taint-based directed whitebox fuzzing. In: 31st International Conference on Software Engineering (ICSE'09), pp.474–484. IEEE (2009)

[72] P. Godefroid, A. Kiezun, M. Y. Levin: Grammar-based whitebox fuzzing. In: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation (PLDI '08), pp. 206–215. ACM (2008)

[73] P. Godefroid, M. Y. Levin, D. Molnar: SAGE: Whitebox fuzzing for security testing, ACM Queue 10, 20-27 (2012)

[74] S. Gorbunov, A. Rosenbloom: AutoFuzz: automated network protocol fuzzing framework. Computer Science and Network Security, 10 pp. 239–245 (2010)

[75] T. Heyman, R. Scandariato, C. Huygens, W. Joosen: Using security patterns to combine security metrics. In: Third International Conference on Availability, Reliability and Security, ARES 08), pp.1156-1163, 4-7 (2008)

[76] M. Howard, J. Pincus, J. M. Wing: Measuring relative attack surfaces. Technical report CMU-CS-03-169, Carnegie Mellon University (2003)

[77] Y. Hsu G. Shu, D. Lee: A model-based approach to security flaw detection of network protocol implementations. In: IEEE International Conference on Network Protocols (ICNP'08). pp. 114–123. IEEE (2008)

[78] International Organization for Standardization: ISO/IEC 15408 – Information technology -- Security techniques – Evaluation criteria for IT security, Version 3.1R3 (2009)

[79] iSixSigma: iSixSigma (2013). [ONLINE] Available at: http://www.isixsigma.com/ [Accessed 25 January 2013]

[80] W. Jansen: Directions in security metrics research. National Institute of Standards and Technology Interagency Report 7564, National Institute of Standards and Technology (2009)

[81] A. Jaquith: Security Metrics: replacing fear, uncertainty, and doubt. Pearson Education (2007)

[82] C. Jing, Z. Wang, X. Shi; X. Yin, J. Wu: Mutation testing of protocol messages based on extended TTCN-3. In: 22nd International Conference on Advanced Information Networking and Applications (AINA'08), pp. 667–674. IEEE (2008)

[83] R. Kaksonen: A functional method for assessing protocol implementation security (licentiate thesis). VTT Publications 448, Technical Research Centre of Finland (2001)

[84] R. Kaksonen, M. Laakso, A. Takanen: System security assessment through specification mutations and fault injection. In: Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security Issues of the New Century, p. 27 (2001)

[85] R. Kaksonen, A. Takanen: Test coverage in model-based fuzz testing. MBT User Conference (2012)

[86] S. Kan: Metrics and models in software quality engineering, second edition. Addison-Wesley Professional (2003)

[87] T. Kitagawa, M. Hanaoka, K. Kono: AspFuzz: A state-aware protocol fuzzer based on application-layer protocols. In: Symposium on Computers and Communications (ISCC'10), pp. 202–208. IEEE (2010)

[88] M. S. Lund, B. Solhaug, K. Stølen: Model-Driven Risk Analysis: The CORAS Approach. Springer (2011)

[89] P. Manadhata, J. M. Wing: An attack surface metric. Technical report CMU-CS-05-155, Carnegie Mellon University (2005)

[90] P. K. Manadhata, K. M. C. Tan, R. A. Maxion, J. M. Wing: An approach to measuring a system's attack surface. Technical report CMU-CS-07-146, Carnegie Mellon University (2007)

[91] E. A. Nichols, G. Peterson: A metrics framework to drive application security improvement, Security & Privacy, 5(2), pp. 88–91 (2007)

[92] C. Patsakis: Measuring security with SecQua (2012). [ONLINE] Available at: http://securitymetrics.org/content/attach/Metricon7.0/Metricon_7_secqua_presentation.pdf [Accessed 10 December 2012]

[93] Open Web Application Security Project: OWASP top ten project (2010). [ONLINE] Available at: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project [Accessed 10 December 2012]

[94] P. Oehlert: Violating assumptions with fuzzing, IEEE Security & Privacy, 3(2), 58–62 (2005)

[95] R. Savola: Towards a security metrics taxonomy for the information and communication technology industry. In: International Conference on Software Engineering Advances (ICSEA 2007)

[96] The Open Web Application Security Project: OWASP Top 10: The ten most critical web application security vulnerabilities, 2007 update (2007). [ONLINE] Available at: https://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf [Accessed 13 January 2013]

[97] R. M. Savola: A security metrics taxonomization model for software-intensive systems. Information Processing Systems 5(4), 197–206 (2009)

[98] R. M. Savola: Towards a risk-driven methodology for privacy metrics development. In: Second International Conference on Social Computing (SocialCom), pp.1086–1092 (2010)

[99] R. M. Savola, C. Frühwirth, A. Pietikäinen: Risk-driven security metrics in agile software development – an industrial pilot study. Journal of Universal Computer Science, 18(12), 1679-1702 (2012)

[100] M. Schneider, J. Großmann, N. Tcholtchev, I. Schieferdecker, A. Pietschker: Behavioral fuzzing operators for UML sequence diagrams. In: 7th Workshop on System Analysis and Modelling 2012 (SAMWkshp'12). LNCS, vol. 7744, pp. 87–103. Springer (2012)

[101] M. Schneider: Model-based behavioural fuzzing. In: Proceedings of the 9th Workshop on Systems Testing and Validation (STV'12), pp. 39–48. Fraunhofer FOKUS (2012) [ONLINE] Available at: http://publica.fraunhofer.de/documents/N-217135.html [Accessed 11 December 2012]

[102] J. Shirk, D. Weinstein, L. Opstad: Fuzzed enough? When it's OK to put the shears away, BlueHat v8 (2008) [ONLINE] Available at: http://technet.microsoft.com/en-us/security/dd285263.aspx [Accessed 11 January 2013]

[103] Ben Smith: Security Test Patterns (2008). [ONLINE] Available at: http://www.securitytestpatterns.org/doku.php [Accessed 25 January 2013]

[104] G. Shu, Y. Hsu, D. Lee: Detecting communication protocol security flaws by formal fuzz testing and machine learning. In: Proceedings of the 28th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems (FORTE'08), pp. 299–304. Springer (2008)

[105] M. Sutton, A. Greene, P. Amini: Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley (2007)

[106] M. Swanson, N. Bartol, J. Sabato, J. Hash, L. Graffo: Security metrics guide for information technology systems. NIST Special Publication 800-55, National Institute of Standards and Technology (2005)

[107] S. Taber, C. Schanes, C. Hlauschek, F. Fankhauser, T. Grechenig: Automated security test approach for SIP-based VoIP softphones. In: Second International Conference on Advances in System Testing and Validation Lifecycle (VALID'10), pp.114–119. IEEE (2010)

[108] A. Takanen, J. DeMott, C. Miller: Fuzzing for Software Security Testing and Quality Assurance. Artech House (2008)

[109] A. Takanen: Fuzzing: the past, the present and the future. In: Actes du 7ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC'09), pp. 202–212 (2009)

[110] R. B. Vaughn, Jr., R. Henning, A. Siraj: Information assurance measures and metrics - state of practice and proposed taxonomy. In: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03). IEEE (2002)

[111] V. Verendel: Quantified security is a weak hypothesis: a critical survey of results and assumptions. In: Proceedings of the 2009 workshop on new security paradigms workshop (NSPW '09), pp. 37–50. ACM (2009)

[112] J. Viide, A. Helin, M. Laakso, P. Pietikäinen, M. Seppänen, K. Halunen, R. Puuperä, J. Röning: Experiences with model inference assisted fuzzing. In: Proceedings of the 2nd conference on USENIX Workshop on offensive technologies (WOOT'08), article no. 2. USENIX Association (2008)

[113] M. Walter, C. Trinitis: Quantifying the security of composed systems. In: Proceeding of Sixth International Conference on Parallel Processing and Applied Mathematics. LNCS, vol. 3911, pp. 1026–1033. Springer (2006)

[114] C. Wang, W. A. Wulf: Towards a framework for security measurement. In: Proceedings of the 20th National Information Systems Security Conference (NISSC'97), pp. 522–533 (1997)

[115] W. Xu, L. Deng, Q. Zhen: Annotating resources in sequence diagrams for testing web security. In: Ninth International Conference on Information Technology: New Generations (ITNG'12), pp.873–874. IEEE (2012)

[116] Y. Yang, H. Zhang, M. Pan, J. Yang, F. He, Z. Li, Z.: A model-based fuzz framework to the security testing of TCG software stack implementations. In: International Conference on Multimedia Information Networking and Security (MINES'09), pp.149–152. IEEE (2009)

[117] I. Schieferdecker, J.Grossmann, M. Schneider, "Model-Based Security Testing," in 7th Workshop on Model-Based Testing, vol. EPTCS 80 (2012)

[118] Shi Yin-sheng, Yuan F. Y. Gu, Tian-yang, "Research on Software Security," Testing World Academy of Science Engineering and Technology, vol. 69 (2010)

[119] Jürgen Doser, Torsten Lodderstedt, David Basin, "Model driven security: From UML models to access control infrastructures," ACM Trans. Soft. Eng. Methodol., vol. 15, pp. 39-91 (2006)

[120] J. Jürjens, Secure Systems Development with UML. Springer (2005)

[121] Z. R. Dai, J. Grabowski, Ø. Haugen, I. Schieferdecker, C. Williams, P. Baker, Model-Driven Testing: Using the UML Testing Profile, 1st edition. Berlin: Springer (2007)

[122] R. Kaksonen, "A functional method for assessing protocol implementation security," VTT Technical Research Center of Finland, 448 (2001)

[123] J. Jürjens, Guido Wimmel, "Specification-Based Test Generation for Security-Critical Systems Using Mutations," in 4th Int. Conf. on Formal Engineering Methods (2002)

[124] E. Wong, D. Xu, L. Wang, "A Threat Model Driven Approach for Security Testing," in Third Int. Workshop on Software Engineering for Secure Systems (2007)

[125] B. K. Aichernig, F. Wotawa, M. Weiglhofer, "Fault-Based Conformance Testing in Practice," Journal of Software and Informatics, vol. 3, pp. 375-411 (2009)

[126] P.-C. Heam, R. Kheddam, F. Dadeau, "Mutation Based Testing of Security Protocols in HLPSL," in Int. Conf. on Sofware Testing Verification and Validation (2011)

[127] J. Oudinet, A. Pretschner, M. Büchler, "Semi-Automatic Security Testing of Web Applications from a Secure Model," in 6th Int. Conf. on Software Security and Reliability, (2012)

[128] L. Fredriksen, B. So Barton, P. Miller, "An Empirical Study of the Reliability of UNIX Utilities," in Workshop of Parallel and Distributed Debugging (1990)

[129] J. DeMott, C. Miller, A. Takanen, "Fuzzing for software security testing and quality assurance," Artech House (2008)

[130] L. L. Pollock, H. Esquivel, B. Hazelwood, S. Ecott S. Sprenkle, "Automated Oracle Comparators for Testing Web Applications," in 18th IEEE Int. Symp. on Software Reliability, (2007)

[131] T. Mouelhi, Y. Le Traon, E. Abgrall, B. Baudry, S. Gombault: Tailored Shielding and Bypass Testing of Web Applications. ICST 2011: 210-219 (2011)

[132] R. Allan Alimi. A Dual-System Approach to Realistic Evaluation of Large-Scale Networked Systems. Ph.D. Dissertation. Yale University, New Haven, CT, USA. (2011)

[133] Cisco Systems. Network Solutions Integrated Test Environment: http://www.cisco.com/en/US/solutions/ns341/ns522/networking_solutions_products_generic content0900aecd80458f98.pdf, 2006.

[134] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In 5th Symposium on Operating System Design and Implementation (OSDI 2002). USENIX Association (2002)

[135] D. Tang, A. Agarwal, D. O'Brien, and M. Meyer. Overlapping Experiment Infrastructure: More, Better, Faster Experimentation. In Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM (2010)

[136] Cisco Systems. Common Routing Problem with OSPF Forwarding Address. http://www.cisco.com/warp/public/104/10.pdf (2005)

[137] D. F. Bacon and S. C. Goldstein. Hardware-assisted Replay of Multiprocessor Programs. In Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging. ACM (1991)

[138] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In Proceedings of the 4th International Conference on Virtual Execution Environments. ACM (2008)

[139] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging for Distributed Applications. In Proceedings of the 2006 USENIX Annual Technical Conference. USENIX (2006)

[140] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical Report dapper-2010-1, Google (2010)

[141] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009. ACM (2009)

[142] Scalable Network Technologies. QualNet Developer. http://www.scalable-networks.com.

[143] G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On Static Reachability Analysis of IP Networks. In INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE (2005)

[144] H. Hamed, E. Al-Shaer, and W. Marrero. Modeling and Verification of IPSec and VPN Security Policies. In 13th IEEE International Conference on Network Protocols (ICNP 2005). IEEE Computer Society (2005)

[145] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In 2nd Symposium on Networked Systems Design and Implementation (NSDI 2005). USENIX Association (2005)

[146] B. Q. Zhao, J. C. Lui, and D.-M. Chiu. Exploring the Optimal Chunk Selection Selection Policy for Data-Driven P2P Streaming Systems. In Proceedings P2P 2009, Ninth International Conference on Peer-to-Peer Computing. IEEE (2009)

[147] A. Bertolino, P. Inverardi, and H. Muccini. Formal methods in testing software architectures. In Formal Methods for Software Architectures: Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures (SFM'03), LNCS 2804, pp. 122–147. Springer (2003)

[148] H. Muccini, M. S. Dias, D. J. Richardson. Towards software architecture-based regression testing. SIGSOFT Softw. Eng. Notes 30(4), 1-7 (2005)

[149] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing Residential Broadband Networks. In Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement 2007. ACM (2007)

[150] K. V. Vishwanath and A. Vahdat. Evaluating Distributed Systems: Does Background Traffic Matter? In Proceedings of the USENIX Annual Technical Conference. USENIX Association (2008)