Robustness Testing Techniques and Tools

Chapter · November 2012

DOI: 10.1007/978-3-642-29032-9_16

6 authors, including:

Zoltan Micskei
Budapest University of Technology and Economics
49 PUBLICATIONS **250** CITATIONS
SEE PROFILE

Henrique Madeira
University of Coimbra
211 PUBLICATIONS **3,748** CITATIONS
SEE PROFILE

Alberto Avritzer
Siemens
86 PUBLICATIONS **1,272** CITATIONS
SEE PROFILE

István Majzik
Budapest University of Technology and Economics
138 PUBLICATIONS **1,724** CITATIONS
SEE PROFILE

Robustness Testing Techniques and Tools

Zoltán Micskei[1] Henrique Madeira[2] Alberto Avritzer[3]
István Majzik[1] Marco Vieira[2]
Nuno Antunes[2]

[1] Budapest Univ. of Technology and Economics, Magyar Tudósok krt. 2, Budapest, Hungary
{majzik, micskeiz}@mit.bme.hu
[2] CISUC, Department of Informatics Engineering, University of Coimbra, Portugal
{henrique, mvieira, nmsa}@dei.uc.pt
[3] Siemens Corporate Research, 755 College Road East, Princeton, NJ, 08540, USA
alberto.avritzer@siemens.com

Abstract. Robustness is an attribute of resilience that measures the behaviour of the system under non-standard conditions. Robustness is defined as the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions. As triggering robustness faults could in the worst case scenario even crash the system, detecting this type of faults is of utmost importance. This chapter presents the state of the art on robustness testing by summarizing the evolution of basic robustness testing techniques, giving an overview of the specific methods and tools developed for major application domains, and introducing penetration testing, a specialization of robustness testing, which searches for security vulnerabilities. Finally, the use of testing results in resilience modelling and analysis is discussed.

Keywords: dependability, security, robustness testing, penetration testing

1 Introduction

Robustness is an attribute of resilience that measures the behaviour of the system under non-standard conditions. Robustness is defined in IEEE Standard 24765:2010 as the degree to which a system operates correctly in the presence of *exceptional inputs* or *stressful environmental conditions* [29]. To further refine the difference between robustness and resilience Avizienis *et al.* defined robustness as "*dependability with respect to external faults, which characterizes a system reaction to a specific class of faults*" [7].

The goal of *robustness testing* is to activate those faults (typically design or programming faults) or vulnerabilities in the system that result in incorrect operation, i.e., robustness failure, affecting the resilience of the system. Robustness testing mostly concentrates on the internal design faults activated through the system interface. The robustness failures are typically classified according to the CRASH criteria [31]: *Catastrophic* (the whole system crashes or reboots), *Restart* (the application has to be restarted), *Abort* (the application terminates abnormally), *Silent* (invalid operation is performed without error signal), and *Hindering* (incorrect error code is returned – note that returning a proper error code is considered as robust

operation). The measure of robustness can be given as the ratio of test cases that exposes robustness faults, or, from the system's point of view, as the number of robustness faults exposed by a given test suite.

Robustness testing can be characterized with the following two components of the tests (stimuli); the *workload* triggers (regular) operation of the system, while the *faultload* contains the exceptional inputs and stressful conditions applied on the system. Depending on how these two loads are balanced, robustness testing can be used for verification and evaluation purposes. Robustness testing can be used as a special kind of conformance testing, where only a faultload is executed against the public interfaces of the system. Overloading can also be considered as a stressful condition, this way stress tests (i.e., submitting only a high amount of workload) are also used to assess the robustness of a system.

Security testing is the process in which the software is verified not for functional purposes but to detect vulnerabilities [42]. A vulnerability is a weakness (an internal fault) that may be exploited by an attacker to cause harm or gain access to the system [58]. *Penetration testing* is a technique to perform security testing and consists of a particular form of robustness testing in which the application execution is analysed when submitted to malicious conditions (i.e., malicious input parameters that try to take advantage of vulnerabilities). Penetration tests are widely used by developers to detect security vulnerabilities in their code [58] and consist of stressing the application from the point of view of an attacker using specific malicious inputs. Penetration testing can be performed manually or automatically. However, automated tools, also referred as web vulnerability scanners, are the typical choice as, comparing to manual tests and inspection, execution time and cost are quite lower. These tools provide an automatic way for searching for vulnerabilities, avoiding the repetitive and tedious task of doing hundreds or even thousands of tests by hand for each vulnerability type.

Robustness benchmarks are agreed-upon and reproducible procedures defined by specifying (1) the workload, (2) the faultload, (3) the standard procedures and rules to execute them, (4) the experimental setup, and (5) the relevant measures that characterize the robustness of the system under benchmarking. As robustness is an attribute of dependability, robustness benchmarks can be considered as a special category of dependability benchmarks.

From the point of view of the lifecycle of systems, on the one hand, robustness testing can be used as an internal step in the development process, complementing the other verification and evaluation activities. On the other hand, robustness tests can be performed (typically by external parties) after the release of the system, to assess its dependability or to compare it to other systems. The detected robustness faults can be handled either by performing corrections in the design and implementation, or by the application of specific wrappers that are able to confine the effects of the faults. This latter solution is particularly relevant when the system contains commercial off-the-shelf (COTS) components, which cannot be modified to correct the faults/robustness weaknesses, thus a wrapper built around the component is the only solution to improve its robustness.

The outline of this chapter is as follows. Section 2 discusses the evolution of basic robustness testing techniques. Section 3 provides an overview of the specific methods and tools developed for major application domains. Section 4 introduces the

penetration testing technique and some of the existing tools. Section 5 briefly discusses the use of testing results for resilience modelling and analysis. Finally, Section 6 concludes the chapter.

2 Robustness Testing

In the past decades many research projects were devoted to the robustness testing of a specific application or application type. The early methods were mainly based on hardware fault injection, but later the research focus moved to software-implemented techniques. In this section we introduce the main milestones, which can be connected to the introduction of new *testing techniques.*

2.1 Injecting physical faults

Early work on robustness testing used *fault injection* (FI) tools to induce or simulate the effects of various hardware related faults. Here a clear distinction shall be made between the purposes of general FI and FI for robustness testing. The general technique assesses the ability of a system or component to handle internal hardware or software faults. In a robustness testing framework, FI can be used to assess the ability of a component to handle interaction faults that are triggered by injecting faults into the environment (e.g., interacting components or underlying layers) while keeping the tested component intact. In this way also the robustness of error detection and error handling mechanisms (considered as components to be tested) can be investigated.

FIAT [12] or FTAPE [60] are examples for FI tools that are reported to be used for such robustness testing purposes.

2.2 Using random inputs

One of the first robustness testing techniques was the generation of *random input* for the system. Random inputs are easy to generate, there is a chance that robustness faults are activated by them, and due to the simple acceptance criteria (crash/hung is checked) there is no need to generate reference output.

Fuzz [46] was one of the first tools supporting this technique. It was utilized in three series of experiments to test reliability and robustness of various applications. In 1990, utility programs on seven variants of Unix operating systems were tested. In 1995, the tests were repeated to check whether robustness of these utilities had been improved and support to test X Window applications were added. Lastly, in 2000, Fuzz was used to test 30 GUI applications on Windows NT. Although the method used was really simple, it detected a great deal of robustness errors, namely 40% of the Unix command line programs and 45% of the Windows NT programs crashed (terminated abnormally) or hung (stopped responding to input within a reasonable length of time) when called with random input data.

Although random testing is a basic technique, it proves to be useful even for modern COTS software systems. The tests in Fuzz were reapplied to MacOS in a

study prepared in 2007 [47] with the following results: 10 command line utilities crashed out of the 135 utilities that were tested (a failure rate of 7%), 20 crashed and 2 hung out of the 30 GUI programs tested (a failure rate of 73%). Thus, it turns out that robustness testing using random inputs is still a viable technique as robustness of common software products has not been significantly improved in general in the last fifteen years.

Fuzzing is extensively applied to security related testing, as presented in a recent book [59].

2.3 Using invalid inputs

Basic software robustness testing technique is the systematic calling of the interface functions of the system under test using parameter values that are selected from the boundaries and outside of their allowed domain. E.g., if allowed values of a parameter are positive integers then robustness tests may contain the zero, a negative number and the MAXINT value.

A typical tool supporting this approach was Riddle [24], which used a grammar-based description of the system's input to generate random and invalid tests. According to the grammar definition legally structured inputs were generated that were filled with random values, possible malicious values (like special or non-printable characters) and boundary values (for example numbers like MAXINT + 1). In this way, syntactically correct inputs could also be created (not just totally random streams), and a greater portion of the systems functionality could be accessed by the tests. Results showed that about 10% of the tests on GNU command line utilities produced unhandled exceptions. The robustness failures observed were mainly memory access violation exceptions, privileged instruction exceptions and illegal instruction exceptions. The typical cause of these exceptions was the improper handling of non-printable characters and (excessively) long input streams.

Another area where invalid inputs can be easily defined and used for robustness testing is low-level OS device drivers. In [44] such drivers were tested by selecting extreme values from the following categories: forbidden value, out of bounds value, invalid pointer assignment, NULL pointer assignment, missing local variable initialization, and missing call of a related function.

2.4 Using type-specific tests

The robustness tests can be further refined by using specific invalid inputs for each function in the system's interface. To minimize the amount of manually created test cases a *type-specific approach* was introduced. The basic idea is that valid and invalid inputs are defined for the data types used in the system's interface functions, and the robustness tests are generated by combining the values for the different parameters. The size of the invalid input domain can be further reduced by using inheritance between the types to test.

The Ballista tool [31] used this approach to compare the robustness of 15 POSIX operating systems using a test suite for 233 function calls. The general goal of the

research was to implement methods to measure the robustness of the exception handling mechanism of systems. The results could be used to evaluate the dependability of a system and characterize how it responds to the failures of other components. In an experiment performed on the Safe Fast I/O (SFIO) library, the performance drawback of robustness hardening was also measured. The tests showed that the performance penalty of proper data validation and parameter checking was fewer than 2%. A good summary of the experiences gained using Ballista can be found in [32].

2.5 Testing object-oriented systems

The type-specific technique mentioned above can be enhanced in object-oriented (OO) systems with the help of automatically building a parameter graph with the type structure. The parameter graph describes how the specific object types used as parameters in method calls can be generated as results of calling constructors or public methods of other classes. This way the generation of an invalid object (needed to test a given method) can be traced back to the call of another method (possibly having parameters of simpler input types).

The JCrasher tool [15] creates robustness tests for Java programs automatically by analyzing which methods could return a type needed for the actual parameters. It examines the type information of the set of Java classes constituting the application and constructs code fragments that will create instances of different types to test the behaviour of public methods with random or invalid data.

In OO applications the testing of exception handling is an important aspect of assessing the robustness of the fault handling and recovery code. Exception flow analysis and testing exception-catch paths is presented in [23].

2.6 Applying mutation techniques

Code mutation techniques [16] can be also applied to generate robustness tests. Starting from a valid code, e.g., a functional test or an application using the system's interfaces, mutation operators can be applied, which resemble the typical faults causing robustness problems (e.g., omitting calls, interchanging calls, replacing normal values in parameters with invalid values).

Mutation and extension of valid test sequences may also help in state-based systems or components to cover more states and transitions than in case of stateless API testing. In [35],first a set of paths is generated to cover state transitions of the tested component, and normal test cases are applied to traverse these paths and bring the component into specific states. In each state, the available methods are called with invalid inputs to test the robustness in that state. This approach is motivated by the fact that complex components may fail differently in different states.

2.7 Model-based robustness testing

The increasingly popular model-driven development paradigm led to the idea of model based testing (using models as formal or semi-formal specification for testing purposes) [13][17] and also model based automated test generation [48]. Naturally, model based test generation can be tailored to create robustness tests by looking for extreme values and conditions on the basis of the pre- and post-conditions, invariants, and constraints fixed in the design model. Model-based testing is currently a very active field of research; here we mention only a few techniques and tools that are relevant to testing robustness.

The first test generation approaches utilized formal specifications and functional models (B, Z, LOTOS, etc.). Constraint-solving techniques were applied to generate boundary values of input domains as well as the corresponding test cases. In state based formalisms, e.g. in IOLTS [19], path searching and model mutation (on the basis of fault models) were applied in order to find tests for concrete robustness criteria. Timed behaviour was modelled and tested using timed automata [21] or extended interoperability models [40].

In case of communication protocols SDL was the primary modelling language used for generating robustness tests [55]. In another work [53], finite state machine models of communication protocols were extended and faulty protocol data units were generated on the basis of a stress operational profile in a statistical approach to model based robustness testing.

In UML based designs the Object Constraint Language (OCL) was used to specify valid domains, this way providing input information also for robustness testing. Typical examples of UML based test generator tools that support (a subset of) OCL were LTG/UML [62] and ParTeG [65].

Model based configuration and execution of robustness testing is complementary to model based test generation. In [51], a framework was presented that fits to the model based development approach by offering to the tester the model of the tested application (using UML class diagram model elements) and domain-specific extensions that allow the configuration of fault injection and robustness testing experiments. The modifications that are required for robustness testing are implemented automatically (using a Java bytecode manipulation technology) on the basis of the model extensions.

2.8 Historical overview of the basic robustness testing approaches

A detailed survey of robustness testing techniques was provided in the context of the ReSIST Network of Excellence, in particular in the report summarizing the state of knowledge [50].

To conclude the section on basic approaches and major supporting tools, Table 1 presents the historical time line of the techniques and evaluates their current applicability, while Fig. 1 presents the relations between the techniques.

Table 1. Historical time line of techniques used for robustness testing.

Testing technique	Introduced	Evaluation of applicability
Injection of physical faults	Early 1990s	Used in specific domains, e.g., in safety-critical systems
Random inputs	Mid 1990s	Basic technique, still useful for off-the-shelf software
Invalid inputs	Late 1990s	Used as part of type-specific testing
Type-specific testing	Around 2000	Very effective technique, used together with mutation
OO approach	Early 2000s	Extension of type-specific tests to OO languages
Mutation techniques	Early 2000s	Effectively complements type-specific techniques

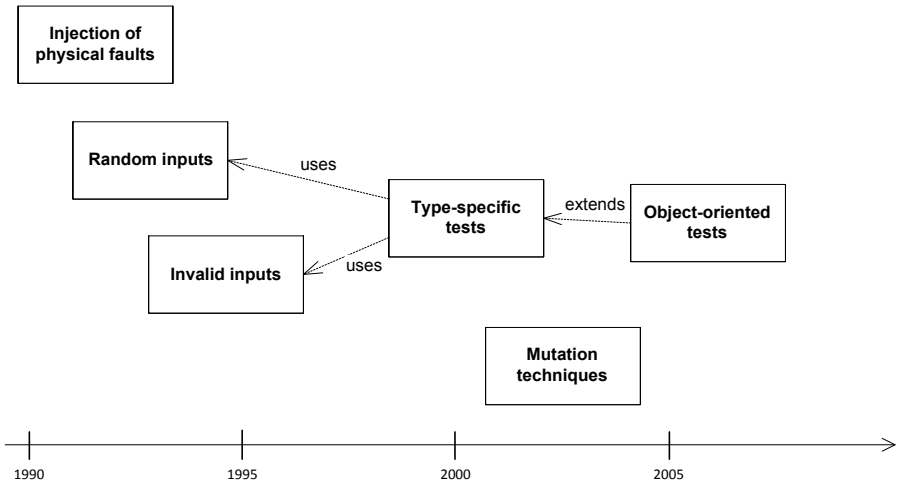

Fig. 1. Relations between the basic robustness testing techniques

3 Robustness testing techniques in specific application domains

In recent years, robustness testing techniques have been successfully used in several application domains. In the following we survey the peculiarities of these techniques.

3.1 User interfaces

State-based testing of graphical user interfaces (GUI) relies on building a graph based model that describes the elements of the interfaces and the connections (e.g., allowed activation sequences) between them. This model is often called an event-flow or event-interaction graph. In a robustness testing scenario the graph describing the normal operation is completed with the connections that are not allowed. The activation of these connections (after reaching the activation of their starting point by a sequence of normal interactions) constitutes the robustness test suite. These robustness tests can be generated automatically based on the possible sequences obtained from the graph model. This technique can be considered as a specific mutation technique that appends an invalid activation after the sequence of valid ones.

The GUITAR framework [43] is a tool set for creating automatic tests for GUIs. It detects the elements of the GUI, e.g., menus, buttons, and constructs an event flow graph. Rapid tests trying to crash the application can be generated from the model after each release of the software. The effectiveness of the method and the tools were demonstrated on the GUI of an open-source office suite.

3.2 High availability middleware

Robustness is a key factor in middleware systems that are applied to provide high availability (HA) services to applications and to manage the configuration of redundant components: robustness faults in the HA middleware can be activated by poor quality application components, and this way one such component may render the whole application inaccessible. The complexity in testing these middleware implementations comes from the highly state-based nature of these systems: without a proper setup code most of the calls in the public interface result in trivial error messages, this way the robustness of the valid operation cannot be tested.

In [45] the common interface specified by the Service Availability Forum was used to test and compare the robustness of different middleware implementations. A robustness test suite was constructed on the basis of the definition of the common interface using a type-specific approach, i.e., various combinations of valid and invalid inputs were generated automatically. To test stressful environmental conditions an operating system call wrapper was implemented, since the manifestation of many stressful conditions (network errors, lack of disk space etc.) can be simulated by injecting errors into the return values of system calls. Additional robustness tests were generated by mutating tests in the functional test suite (changing the order of calls, applying invalid parameter values when a given state is reached etc.). These methods exposed distinct robustness faults in the system, demonstrating the importance of combining the different testing techniques.

3.3 Real-time executives

Microkernels are currently common components in a wide range of applications, ranging from daily-use appliances (e.g., mobile devices) to space-borne vehicles.

Robustness testing of microkernels has been addressed using stressful operational conditions, invalid inputs at the public interfaces, and fault injection.

In [5] the MAFALDA tool is used to collect objective failure data and to find ways to improve the error detection capabilities of the Chorus and the Lynx microkernels. A specialized version of that tool, the MAFALDA-RT, is used in [54] to test real-time systems. The tool applies a novel method to cope with the problem of temporal intrusiveness caused by the use of software implemented fault injection (SWIFI) tools. In addition to detecting typical failure modes (e.g., application hang, system hang, exception, etc.), the observation capabilities of MAFALDA-RT are extended to consider temporal properties characterizing both the executive and application layers (e.g., task processing, task synchronization, context-switch, system calls, etc.).

The Xception tool [14] is a SWIFI tool that was used in several experiments of microkernel robustness testing (e.g., [38], [11] and [49]). This tool is able to introduce perturbations in the processor to emulate errors at the hardware and software level (in the case of a software fault, the fault is introduced when user applications are being executed). The robustness evaluation consists of assessing the abilities of the microkernel to handle the stressful situation caused by the existence of faults. One interesting work used Xception to emulate hardware transient faults and observe their effects at the user application and at the operating system level [37]. The results demonstrated that errors occurring in one user-mode application can propagate to other user-mode applications through the OS itself.

3.4 OLTP and DBMS

Although not as common as other types of systems, Online Transaction Processing (OLTP) systems were also used in past robustness-related works. The following two studies are worth noting.

The work presented in [64] presents a dependability benchmark (covering also the attributes of robustness) for OLTP systems. The experiments exposed the OLTP systems under observation to stressful conditions and measured the performance penalty of the Database Management System (DBMS) engine and the rate of occurrence of data corruption on the data tables. The stressful conditions were caused by the injection of operator errors. This work demonstrated that it is possible to assess the dependability and robustness properties of OLTP systems and rank the systems under study according to the results.

The work in [20] presents a method to detect intrusions and malicious data access based on DBMS auditing. Although this work is not exactly inline with the traditional robustness testing described earlier in this section, assessing the ability to prevent or detect intrusion can constitute a measure to the robustness of a system (in this case, the security attributes of that system).

3.5 Web Services

Web Services are a widespread technology to implement services accessible over a computer network. Each Web Service has a well-defined interface (usually using a

standardized description language). Several methods were proposed to generate robustness tests based on these interfaces for Web Services.

The work described in [61] presents a specification-based robustness testing framework for Web Services. The testing framework includes the analysis of the service specification to ensure that it is complete and consistent. It applies the Covering Scenario Generation algorithm to identify the locations where incompleteness and inconsistency exist. The testing framework also includes the robustness testing of the Web Service by generating positive test cases (that should be successful) as well as negative test cases (that should not be successful).

WebSob [39] is a tool that automates the generation and execution of test cases for Web Services. This tool executes Web Service requests using the provider's Web Service Description Language (WSDL) specification. This tool was applied to freely available implementation of Web Services and revealed several robustness problems. The tool does not require the knowledge of the implementation of the Web Services under test.

The work presented in [63] defines a benchmark to assess the robustness of Web Services. It uses invalid data in the method invocations to discover both programming and design errors. The parameter values used in the method invocations are modified (i.e., corrupted) based on the data types and the semantics of the method parameters. The Web Services are classified according to the type and number of failures observed during the tests.

The WS-FIT tool [36] performs a dependability analysis of Web Services. For the sake of robustness testing it performs network fault injection by capturing, modifying, and retransmitting SOAP messages. This technique allows for easy corruption of RPC method invocation within SOAP messages and can emulate the following errors: corruption of input and output data, omission or duplication of messages, delay of messages.

4 Penetration Testing

Penetration testing, a specialization of robustness testing, consists of the analysis of the program execution in the presence of malicious inputs (based on a database of know malicious values for specific applications or generated based on predefined rules), searching for potential vulnerabilities [58]. Hackers are nowadays moving their focus to this kind of vulnerabilities and explore applications' inputs with specially tampered values trying to find weaknesses. These vulnerabilities cannot be mitigated by traditional security mechanisms such as firewalls and intrusion detection system, thus highlighting the importance of detecting these vulnerabilities before deployment. Also, the exposition of web applications makes them particularly prone to attacks that try to exploit code vulnerabilities. In this section we overview existing techniques, focusing particularly in web applications and code vulnerabilities, as penetration testing is particularly important in these scenarios. Penetration testing is, in fact, the most common approach to detect vulnerabilities in web applications and web applications are the most frequent context in which penetration testing is used.

4.1 "Black-box" penetration testing

In "black-box" penetration testing the tester does not know the internals of the application and it uses fuzzing techniques over the applications requests [58]. The tester needs no knowledge of the implementation details and tests the inputs of the application from the user's point of view by applying malicious inputs. The number of tests can reach hundreds or even thousands for each vulnerability type. The vulnerability detection is based essentially on the analysis of the application output. The human tester or the tool analyses the contents of this output including values or errors returned and exceptions raised. The vulnerabilities are detected when certain patterns found in the response are caused by the attacks launched. Many black-box penetration testing techniques were proposed in the past. We introduce a few in the next paragraphs due to the relevant innovations they introduced.

WAVES [27] is a black-box technique for testing web applications for SQL-Injection vulnerabilities. The technique is based on a reverse engineering process that identifies the data entry points of a Web application and attacks them with malicious patterns. An algorithm is proposed to allow "deep injection" and to eliminate false negatives. During the attack phase, the application's responses to the attacks are monitored and machine-learning techniques are used to improve the attack methodology.

SecuBat [30] is an open-source penetration testing tool that uses a black-box approach to crawl and scan web sites for the presence of exploitable SQL injection and cross-site scripting (XSS) vulnerabilities. SecuBat does not rely on a database of known bugs. Instead, it tries to exploit the distinctive properties of application-level vulnerabilities. To increase the confidence in the correctness of the results, the tool also attempts to automatically generate proof-of-concept exploits in certain cases.

A black-box taint-inference technique for the detection of injection attacks is proposed in [56]. The technique does not require any intrusive source-code or binary instrumentation of the application to be protected; instead, it intercepts the inputs and outputs of the application. Then, the technique infers tainted data in the intercepted SQL statements, and then employs syntax and taint-aware policies to detect unintended use of tainted data.

In [41] an automated penetration testing tool is presented that can find reflected and stored XSS vulnerabilities in web applications. The proposed technique improves the effectiveness of penetration testing by leveraging input from real users as a starting point for its testing activity. The technique follows an entire user's session using recorded real user inputs to generate test cases to launch fuzzing attacks. This way, the technique increases the code coverage by exploring pages that are not reachable for other tools. The experiments show that the approach is able to test more thoroughly the web applications and identify more bugs than a number of open-source and commercial tools.

A vulnerability scanner for web services that performs better than the commercial ones currently available is presented in [4]. The work focuses on the detection of SQL Injection vulnerabilities, one of the most common and most critical types of vulnerabilities in web environments. The proposed approach is based on a large set of attacks and includes an enhanced response analysis (i.e., vulnerability detection) algorithm. Experimental evaluation shows that the proposed approach performs much

better than well-known commercial tools, achieving very high detection coverage while maintaining the false positives rate quite low.

4.2 "Gray-box" penetration testing

The main limitation of black-box approaches is that vulnerability detection is limited by the output of the tested application. Gray-box approaches consist of complementing black-box testing with white-box techniques to overcome such limitation.

Dynamic program analysis is based on the analysis of the behaviour of the software while executing it [58]. The idea is that by analysing the internal behaviour of the code in the presence of realistic inputs it is possible to identify bugs and vulnerabilities. Obviously, the effectiveness of dynamic analysis depends strongly on the input values (similarly to black-box testing), but it takes advantage of the observation of the source code. For improving the effectiveness of dynamic program analysis, the program must be executed with sufficient test inputs. Code coverage analysers help guaranteeing an adequate coverage of the source code [18, 6].

Source code or bytecode instrumentation is a technique that can be used to develop runtime anomaly detection tools. In [34] it is proposed an anomaly detection approach to secure web services against SQL and XPath Injection attacks. The web service is instrumented in such way that all the SQL/XPath commands used are intercepted before being issued to the data source. The approach consists of two phases. First, in the learning phase, the approach learns the regular patterns of the queries being issued. Then, at runtime, the commands are compared with the patterns learned previously in order to detect and abort potentially harmful requests. The problem of this technique is that it does not include the generation of the requests to use during the learning phase and so it requires the user to exercise the web service in the learning phase.

In [3] the runtime anomaly detection approach from [34] was enhanced with an automated workload and attackload generation technique in order to be able to detect SQL and XPath Injection vulnerabilities in web services. This way, after the instrumentation of the web service a workload is generated using information about the domains of the parameters of the web service operations. Learning takes place while executing the workload to exercise the web service. Afterwards, an attackload is generated and used to attack the web service. Vulnerabilities are detected by comparing the incoming commands during attacks with the valid set of commands previously learned.

"*Acunetix AcuSensor Technology*" [2] is a technique introduced by Acunetix that combines black-box scanning with feedback obtained during the test execution. This feedback is provided by sensors previous placed, using code instrumentation, inside the source code or bytecode. Using this technique it is possible to find more vulnerabilities, to indicate in the code exactly where they are, and to report less false positives. This technology is available for web applications, specifically .NET and PHP web applications. In case of .NET this technology can be injected in bytecode.

Two techniques that combine static and dynamic analysis have been proposed to perform automated test generation to find SQL Injection vulnerabilities.

SQLUnitGen, presented in [57], is a tool that combines static analysis with unit testing to detect SQL injection vulnerabilities. The tool uses a third-party test case generator and then modifies the test cases to introduce SQL injection attacks. In practice, these attacks are obtained by using static analysis to trace the flow of user input values to the point of query generation. Sania, presented in [33], is a testing framework to detect SQL Injection vulnerabilities in web applications during development and debugging phases. Sania intercepts the SQL queries between a web application and a database and constructs parse trees of these queries. Terminal leafs of parse trees typically represent vulnerable spots. The technique then generates attacks according to the syntax and semantics of these potentially vulnerable spots. Finally, Sania compares the parse trees of the original SQL query with the ones resulting after an attack to assess the safety of these spots. The differences between the parse trees are considered vulnerabilities, originating a warning.

Runtime anomaly detection tools are used as attack detection systems to protect the applications at runtime. However, a detected attack is typically the exploitation of an existing vulnerability, therefore showing that at least a vulnerability exists. Examples include [34] and AMNESIA (Analysis and Monitoring for NEutralizing SQL-Injection Attacks) [25]. AMNESIA combines static analysis and runtime monitoring to detect and avoid SQL injection attacks. Static analysis is used to analyse the source code of a given web application building a model of the legitimate queries that such application can generate. At runtime, AMNESIA monitors all dynamically generated queries and checks them for compliance with the statically generated model. When a query that violates the model is detected it is classified as an attack and is prevented from accessing the database. The problem is that the model built during the static code analysis may be incomplete and unrealistic because it lacks a dynamic view of the runtime behaviour of the application.

While other works focused on identifying vulnerabilities related to the use of external inputs without sanitizations, the work presented in [10] introduces an approach that combines static and dynamic analysis techniques to analyse the correctness of sanitization processes in web applications. First, a technique based on static analysis models the modifications that the inputs suffer along the code paths. This approach uses a conservative model of string operations, which might lead to false positives. Then, a second technique based on dynamic analysis works bottom-up from the sinks and reconstructs the code used by the application to modify the inputs. The code is then executed, using a large set of malicious input values to identify exploitable flaws in the sanitization process.

4.3 Examples of penetration testing tools

Penetration testing tools provide an automatic way to search for vulnerabilities avoiding the repetitive and tedious task of doing hundreds or even thousands of tests by hand for each vulnerability type. The most common automated security testing tools used in web applications are generally referred to as web security scanners (or web vulnerability scanners). Web security scanners are often regarded as an easy way to test applications against vulnerabilities. These scanners have a predefined set of tests cases that are adapted to the application to be tested. In practice, the user only

needs to configure the scanner and let it test the application. Once the test is completed the scanner reports existing vulnerabilities (if any detected). Most of these scanners are commercial tools, but there are also some free application scanners often with limited use, since they lack most of the functionalities of their commercial counterparts.

Two very popular free security scanners that support web services testing are Foundstone WSDigger [22] and WSFuzzer [52]. WSDigger is a free open source tool developed by Foundstone that executes automated penetration testing in web services. Only one version of this software was released up to now (in December 2005). The tool contains sample attack plug-ins for SQL Injection, cross-site scripting (XSS), and XPath Injection, but it was released as open-source to encourage users to develop and share their own plug-ins and its test files are simple to edit to add new test cases. WSFuzzer is a free open source program that mainly targets HTTP based SOAP services. This tool was created based on real-world manual SOAP penetration testing work, automating it. Nevertheless, the tool is not meant to replace a solid manual human analysis. One problem of this tool is that its configuration is very complex. The main problem of both WSDigger and WSFuzzer is that, in fact, they do not detect vulnerabilities: they attack the web service under testing and log the responses leaving to the user the task of examining those logs and identify the vulnerabilities. This requires the user to be an "expert" in security and to spend a huge amount of time to examine all the results.

As for commercial scanners, three brands currently lead the market: HP WebInspect [26], IBM Rational AppScan [28] and Acunetix Web Vulnerability Scanner [1].

HP WebInspect is a tool that "*performs web application security testing and assessment for today's complex web applications, built on emerging Web 2.0 technologies. HP WebInspect delivers fast scanning capabilities, broad security assessment coverage and accurate web application security scanning results*" [26]. This tool includes pioneering assessment technology, including simultaneous crawl and audit (SCA) and concurrent application scanning. It is a broad application that can be applied for penetration testing in web-based applications.

IBM Rational AppScan "*is a leading suite of automated Web application security and compliance assessment tools that scan for common application vulnerabilities*" [28]. This tool is suitable for users ranging from non-security experts to advanced users that can develop extensions for customized scanning environments. IBM Rational AppScan can be used for penetration testing in web applications, including web services.

Acunetix Web Vulnerability Scanner "*is an automated web application security testing tool that audits a web applications by checking for exploitable hacking vulnerabilities*" [1]. Acunetix WVS can be used to execute penetration testing in web applications or web services and is quite simple to use and configure. The tool includes numerous innovative features, for instance the "AcuSensor Technology" [2].

5 Resilience Modelling and Analysis using Testing Results

The Chapter on performance testing contains an overview of the application of modelling and analysis based in performance testing results. The described approach can be generalized to robustness testing with proper metrics and model state definitions.

The approach consists of defining a resilience related metric that can be derived from the system security, reliability or performance requirements as follows:

"*The fraction of time the system satisfies the defined resilience requirements specifications during an observation period (0,t)*"

The steps required to implement the approach presented in [8] are:

1. A resilience-based state definition needs to be devised. The resilience-based state definition maps system resources to the events the system is designed to be resilient to. For example, resilience related events could be faults, security intrusions, or any other system activity that needs to be modelled.
2. Resilience modelling using the approach introduced in [9, 8] requires the definition of Markov chains to contain the states and associated events. For example, a failure-based Markov chain captures failures and repair events. Detailed descriptions of the approaches with examples are presented in [9, 8].
3. The utilization of testing results pass/fail conditions to assess the resilience metric requires the association of each resilience test with a Markov chain state. All states associated with a test case result pass condition are used in the resilience metric computation. In addition, the solution of the Markov chain enables the association of the resilience metric with a notion of system reliability.

As topics for future research, we foresee a probabilistic quantification of robustness related initiation and completion events (e.g. failure/repair) in some of the relevant non-functional requirement domains, such as performance, reliability and security.

6 Conclusion

This chapter presented an overview on robustness testing techniques, providing examples of applications to several domains. In particular, we have introduced penetration testing, where black-box and gray-box tests are used for detecting security vulnerabilities. Finally, we have also introduced a quantitative approach for the evaluation of a robustness metric by using robustness testing results. The approach is based on the definition of a system model in terms of robustness, the definition of test cases that are related to the model states, and assessing pass/fail for each test case executed in the robustness testing phase.

The robustness testing approaches presented in this chapter can be used to define a systematic process that includes robustness metric definition, modelling of system

robustness, robustness test cases generation, automated tools for robustness testing, and the assessment of the system robustness metric by using the pass/fail robustness test case results. The current state of the art in robustness testing emphasizes the need for additional studies on the identification of the most useful robustness models, and the associated probabilistic quantification of the robustness states that are visited by failures and security penetration events.

Acknowledgments

The work presented in this chapter was partially funded by the European Commission under project AMBER - Assessing, Measuring and Benchmarking Resilience, IST - 216295, funded by the European Union, 2009.

References

1. Acunetix: Acunetix Web Vulnerability Scanner, (2011) http://www.acunetix.com/vulnerability-scanner/
2. Acunetix: AcuSensor Technology, (2011) http://www.acunetix.com/vulnerability-scanner/acusensor.htm
3. Antunes, N. et al.: Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services. IEEE International Conference on Services Computing. pp. 260-267, Bangalore, India (2009) doi:10.1109/SCC.2009.23
4. Antunes, N., Vieira, M.: Detecting SQL Injection Vulnerabilities in Web Services. Fourth Latin-American Symposium on Dependable Computing. pp. 17-24, IEEE, Joao Pessoa, Brazil (2009) doi:10.1109/LADC.2009.21
5. Arlat, J. et al.: Dependability of COTS microkernel-based systems. IEEE Transactions on Computers. 51, 2, pp. 138-163 (2002) doi:10.1109/12.980005
6. Atlassian: Clover - Code Coverage for Java, (2011) http://www.atlassian.com/software/clover/
7. Avižienis, A. et al.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing. 1, 1, pp. 11-33 (2004) doi:10.1109/TDSC.2004.2
8. Avritzer, A. et al.: Automated generation of test cases using a performability model. IET Software. 5, 2, pp. 113-119 (2011) doi:10.1049/iet-sen.2010.0035
9. Avritzer, A., Weyuker, E.R.: The automatic generation of load test suites and the assessment ofthe resulting software. IEEE Transactions on Software Engineering. 21, 9, pp. 705-716 (1995) doi:10.1109/32.464549
10. Balzarotti, D. et al.: Saner: Composing static and dynamic analysis to validate sanitization in web applications. IEEE Symposium on Security and Privacy. pp. 387-401 (2008) doi:10.1109/SP.2008.22
11. Barbosa, R. et al.: Verification and Validation of (Real Time) COTS Products using Fault Injection Techniques. Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems, 2007. ICCBSS '07. pp. 233-242 IEEE (2007) doi: 10.1109/ICCBSS.2007.45

12. Barton, J.H. et al.: Fault injection experiments using FIAT. IEEE Transactions on Computers. 39, 4, pp. 575-582 (1990) doi: 10.1109/12.54853
13. Broy, M. et al.: Model-Based Testing of Reactive Systems: Advanced Lectures. Lecture Notes in Computer Science. (2005)
14. Costa, D. et al.: Xception™: A Software Implemented Fault Injection Tool. In: Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation. pp. 125-139 (2004) doi:10.1007/0-306-48711-X_8
15. Csallner, C., Smaragdakis, Y.: JCrasher: an automatic robustness tester for Java. Software: Practice and Experience. 34, 11, pp. 1025–1050 (2004) doi:10.1002/spe.602
16. DeMillo, R.A. et al.: An extended overview of the Mothra software testing environment. Proc. of the Second Workshop on Software Testing, Verification, and Analysis, pp. 142-151 IEEE (1988) doi:10.1109/WST.1988.5369
17. Dias Neto, A.C. et al.: A survey on model-based testing approaches: a systematic review. Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies (WEASELTech '07). pp. 31-36 (2007) doi:10.1145/1353673.1353681
18. Doliner, M.: Cobertura, http://cobertura.sourceforge.net/
19. Fernandez, J.-C. et al.: A Model-Based Approach for Robustness Testing. In: Khendek, F. and Dssouli, R. (eds.) Testing of Communicating Systems. pp. 333-348, Springer (2005) doi:10.1007/11430230_23
20. Fonseca, J. et al.: Online detection of malicious data access using DBMS auditing. Proceedings of the 2008 ACM symposium on Applied Computing. pp. 1013–1020 ACM Press (2008) doi:10.1145/1363686.1363921
21. Fouchal, H. et al.: Robustness testing of composed real-time systems. Journal of Computational Methods in Science and Engineering. 10, pp. 135-148 (2010)
22. Foundstone, Inc.: Foundstone WSDigger, (2011) http://www.foundstone.com/us/resources/proddesc/wsdigger.htm
23. Fu, C. et al.: Robustness testing of Java server applications. IEEE Transactions on Software Engineering. 31, 4, pp. 292-311 (2005) doi:10.1109/TSE.2005.51
24. Ghosh, A.K., Schmid, M.: An approach to testing COTS software for robustness to operating system exceptions and errors. Proc. of 10th Int. Symp. on Software Reliability Engineering. pp. 166-174 (1999) doi:10.1109/ISSRE.1999.809321
25. Halfond, W.G.., Orso, A.: AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. Proc. of the 20th IEEE/ACM Int. Conf. on Automated software engineering. pp. 174-183 (2005) doi:10.1145/1101908.1101935
26. Hewlett-Packard Development Company: HP WebInspect, (2011) https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-201-200%5E9570_4000_100__
27. Huang, Y.-W. et al.: Web application security assessment by fault injection and behavior monitoring. Proceedings of the 12th international conference on World Wide Web. pp. 148-159 ACM, (2003) doi:10.1145/775152.775174
28. International Business Machines: IBM Rational AppScan, (2011) http://www-01.ibm.com/software/awdtools/appscan/
29. IEEE: Systems and software engineering - vocabulary, Standard 24765:2010, (2010) doi:10.1109/IEEESTD.2010.5733835

30. Kals, S. et al.: Secubat: a web vulnerability scanner. Proc. of the 15th int. conf. on World Wide Web. pp. 247-256 (2006) doi:10.1145/1135777.1135817
31. Koopman, P., DeVale, J.: The exception handling effectiveness of POSIX operating systems. IEEE Transactions on Software Engineering. 26, 9, pp. 837-848 (2000) doi:10.1109/32.877845
32. Koopman, P. et al.: Interface Robustness Testing: Experience and Lessons Learned from the Ballista Project. Dependability Benchmarking for Computer Systems. Wiley-IEEE CS Press (2008) doi:10.1002/9780470370506.ch11
33. Kosuga, Y. et al.: Sania: Syntactic and semantic analysis for automated testing against SQL injection. 23rd Annual Computer Security Applications Conference. IEEE Computer Society. pp. 107–117 (2007) doi:10.1109/ACSAC.2007.20
34. Laranjeiro, N. et al.: Protecting Database Centric Web Services against SQL/XPath Injection Attacks. Database and Expert Systems Applications. pp. 271–278 (2009) doi:10.1007/978-3-642-03573-9_22
35. Lei, B. et al.: Robustness testing for software components. Science of Computer Programming. 75, 10, pp. 879-897 (2010) doi:10.1016/j.scico.2010.02.005
36. Looker, N. et al.: WS-FIT: a tool for dependability analysis of Web services. Proc. of the 28th Annual Int. Computer Software and Applications Conference, COMPSAC 2004. pp. 120-123 (2004) doi:10.1109/CMPSAC.2004.1342690
37. Madeira, H. et al.: The OLAP and data warehousing approaches for analysis and sharing of results from dependability evaluation experiments. Proceedings of 2003 International Conference on Dependable Systems and Networks, 2003. pp. 86-91 IEEE (2003) doi:10.1109/DSN.2003.1209920
38. Maia, R. et al.: Xception fault injection and robustness testing framework: a case-study of testing RTEMS. VI Test and Fault Tolerance Workshop (jointly organized with the 23rd Brazilian Symposium on Computer Networks (SBRC)). (2005)
39. Martin, E. et al.: WebSob: A Tool for Robustness Testing of Web Services. 29th International Conference on Software Engineering - Companion, 2007. ICSE 2007 Companion. pp. 65-66 (2007) doi:10.1109/ICSECOMPANION.2007.84
40. Mattiello-Francisco, F. et al.: Extended interoperability models for timed system robustness testing. IEEE Latin-American Conference on Communications, LATINCOM '09. pp. 1-6 IEEE (2009) doi:10.1109/LATINCOM.2009.5304903
41. McAllister, S. et al.: Leveraging User Interactions for In-Depth Testing of Web Applications. Recent Advances in Intrusion Detection. pp. 191-210 (2008) doi:10.1007/978-3-540-87403-4_11
42. McGraw, G., Potter, B.: Software Security Testing, IEEE Security and Privacy, 2:81-85 (2004) doi:10.1109/MSP.2004.84
43. Memon, A.M.: An event-flow model of GUI-based applications for testing. Software Testing, Verification and Reliability. 17, 3, pp. 137-157 (2007) doi:10.1002/stvr.v17:3
44. Mendonca, M., Neves, N.: Robustness Testing of the Windows DDK. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN '07. pp. 554-564 IEEE (2007) doi:10.1109/DSN.2007.85
45. Micskei, Z. et al.: Comparing Robustness of AIS-Based Middleware Implementations. In: Malek, M. et al. (eds.) Service Availability. pp. 20-30 Springer (2007) doi:10.1007/978-3-540-72736-1_3

46. Miller, B. et al.: Fuzz revisited: A re-examination of the reliability of Unix utilities and services. Computer Sciences Technical Report #1268, University of Wisconsin-Madison. (1995)
47. Miller, B.P. et al.: An empirical study of the robustness of MacOS applications using random testing. ACM SIGOPS Operating Systems Review. 41, pp. 78-86 (2007) doi:10.1145/1228291.1228308
48. MOGENTES Consortium: State of the Art Survey - Part A: Model-based Test Case Generation, (2008) https://www.mogentes.eu/
49. Moraes, R. et al.: Experimental Risk Assessment and Comparison Using Software Fault Injection. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN '07. pp. 512-521 IEEE (2007) doi:10.1109/DSN.2007.45
50. RESIST NoE: Resilience-Building Technologies: State of Knowledge. Deliverable D12, (2006) http://www. resist-noe.org/outcomes/outcomes.html
51. Olah, J., Majzik, I.: A Model Based Framework for Specifying and Executing Fault Injection Experiments. Fourth International Conference on Dependability of Computer Systems, DepCos-RELCOMEX '09. pp. 107-114 IEEE (2009) doi:10.1109/DepCoS-RELCOMEX.2009.41
52. OWASP Foundation: OWASP WSFuzzer Project, (2011) http://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project
53. Popovic, M., Kovacevic, J.: A Statistical Approach to Model-Based Robustness Testing. 14th Annual IEEE Int. Conf. and Workshops on the Engineering of Computer-Based Systems, pp. 485-494 (2007) doi:10.1109/ECBS.2007.13
54. Rodriguez, M. et al.: MAFALDA-RT: a tool for dependability assessment of real-time systems. Proceedings of International Conference on Dependable Systems and Networks, pp. 267-272 (2002) doi:10.1109/DSN.2002.1028909
55. Saad-Khorchef, F. et al.: A framework and a tool for robustness testing of communicating software. Proceedings of the 2007 ACM symposium on Applied computing. SAC '07. pp. 1461-1466, (2007) doi:10.1145/1244002.1244315
56. Sekar, R.: An efficient black-box technique for defeating web application attacks. Proceedings of the 16th Annual Network and Distributed System Security Symposium. (2009)
57. Shin, Y. et al.: SQLUnitGen: SQL Injection Testing Using Static and Dynamic Analysis. Proceedings of the 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006). Raleigh, NC, USA (2006)
58. Stuttard, D., Pinto, M.: The web application hacker's handbook: discovering and exploiting security flaws. Wiley Publishing, Inc. (2007)
59. Takanen, A. et al.: Fuzzing for software security testing and quality assurance. Artech House, Norwood, MA (2008)
60. Tsai, T.K. et al.: An approach towards benchmarking of fault-tolerant commercialsystems. Proceedings of Annual Symposium on Fault Tolerant Computing, 1996. pp. 314-323 IEEE (1996) doi:10.1109/FTCS.1996.534616
61. Tsai, W.T. et al.: A robust testing framework for verifying Web services by completeness and consistency analysis. IEEE Int. Workshop on Service-Oriented System Engineering, pp. 151- 158 (2005) doi:10.1109/SOSE.2005.2
62. Utting, M., Legeard, B.: Practical model-based testing: a tools approach. Morgan Kaufmann (2007)

63. Vieira, M. et al.: Benchmarking the robustness of web services. 13th Pacific Rim International Symposium on Dependable Computing, PRDC 2007. pp. 322–329 (2007) doi:10.1109/PRDC.2007.24
64. Vieira, M., Madeira, H.: A dependability benchmark for OLTP application environments. Proceedings of the 29th international conference on Very large data bases-Volume 29. pp. 743-753 (2003)
65. Weißleder, S., Schlingloff, B.-H.: Deriving Input Partitions from UML Models for Automatic Test Generation. In: Giese, H. (ed.) Models in Software Engineering. pp. 151-163 (2008) doi: 10.1007/978-3-540-69073-3_17