

Improving Error Detection Rate Using Retesting in Automated Security Testing Tools

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Karin Kernegger BSc

Matrikelnummer 0625123

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Dr.techn. Martin Kampel

Mitwirkung: Dipl.-Ing. Dr.techn. Christian Schanes

Wien, 16.08.2013

(Unterschrift Verfasserin)

(Unterschrift Betreuung)

Improving Error Detection Rate Using Retesting in Automated Security Testing Tools

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Business Informatics

by

Karin Kernegger BSc

Registration Number 0625123

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Dipl.-Ing. Dr.techn. Martin Kampel
Assistance: Dipl.-Ing. Dr.techn. Christian Schanes

Vienna, 16.08.2013

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Karin Kernegger BSc
Römerweg 300, 2722 Winzendorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift Verfasserin)

Security is a process.

Bruce Schneier

Contents

1	Introduction	5
1.1	Problem definition	5
1.2	Expected results	5
1.3	Structure of this thesis	6
2	Introduction to software security	7
2.1	Importance and problems of software security	7
2.2	Terminology concerning attacks	7
2.2.1	Bug	8
2.2.2	Flaw	8
2.2.3	Vulnerability	8
2.3	Properties of secure software	8
2.3.1	CIA-triangle	8
2.3.2	Parkerian Hexad	9
3	Software security testing	11
3.1	Introduction to software testing	11
3.1.1	Software testing methods	11
3.1.2	Types of software tests	12
3.2	Software testing process	13
3.2.1	Planning and control	13
3.2.2	Analysis and design	14
3.2.3	Implementation and execution	14
3.2.4	Evaluating exit criteria and reporting	15
3.2.5	Test closure activities	15
3.3	Security testing	16
3.3.1	Security development lifecycle	16
3.3.2	Abuse cases	18
3.3.3	Security requirements	19
3.3.4	Architectural risk analysis	19
3.3.5	Risk-based security tests	19
3.3.6	Code reviews	19
3.3.7	Penetration testing	20

3.3.8	Security operations	20
3.4	Test automation to reveal security vulnerabilities	21
3.4.1	Introduction to test automation in software testing	21
3.4.2	Advantages of test automation for revealing vulnerabilities	21
3.4.3	Static code analysis	22
3.4.4	Introduction to fuzzing	22
3.4.5	State of the Art of fuzzing	23
3.4.6	Area of operation for fuzzing	24
3.4.7	Kinds of fuzzers	25
3.4.8	Architecture of a fuzzer	25
3.4.9	Description of steps for testing with a fuzzer	27
3.5	Types of software vulnerabilities	29
3.5.1	Injection attacks	29
3.5.2	Cross-Site-Scripting (XSS)	30
3.5.3	Denial of Service (DOS)	32
3.5.4	Directory traversal	32
4	Architecture of the security testing framework	33
4.1	Description of the automated security testing framework	33
4.1.1	Architecture of the framework	33
4.1.2	Analyzer	37
4.2	Problem with detection of security errors	37
4.3	Monitoring the system under test	37
4.3.1	Graphical User Interface of the SUT	39
4.3.2	Systemload	39
4.3.3	Logfile	39
4.3.4	Network traffic	40
4.3.5	TCP/UDP reachability	41
4.3.6	Opened files	41
4.3.7	Response message of the SUT	41
4.3.8	Response time of the SUT	41
4.3.9	Spawned processes	42
4.3.10	Database access	42
4.3.11	Microsoft Windows Registry	42
4.3.12	Output of a debugger	42
4.4	Attack detection by automatic learning of IDS	43
4.4.1	Methods for intrusion detection	43
4.4.2	Anomaly detection	44
4.4.3	Machine learning concepts	47
4.4.4	Concepts of abnormal behavior checking	48
4.4.5	Summary	49
4.5	Automatic learning for security tests	49
4.5.1	Valid cases	50

4.5.2	Learning phase	50
4.5.3	Fuzzing phase	50
4.6	Data Mining and Text Mining	51
4.6.1	Functionalities of Data Mining	51
4.6.2	Functionalities of Data Mining for the fuzzer	53
4.6.3	Concepts of Text Mining	54
4.6.4	Concepts of Text Mining for the fuzzer	56
5	Retesting concept	59
5.1	Introduction into retesting of test cases for security tests	60
5.2	Retesting concept	61
5.2.1	General description of the retesting concept	61
5.2.2	Retesting the different metrics of the fuzzer	64
5.2.3	Vulnerability findings with retesting	71
6	Proof of concept and evaluation of the retesting concept	72
6.1	Proof of concept implementation in the fuzzing framework	72
6.1.1	Monitoring the SUT and indications of security errors	72
6.1.2	Automatic learning, value comparison and Text Mining	72
6.1.3	Retesting	75
6.2	Results using the retesting concept for a sample application	75
6.2.1	Retesting for detecting SQL injection vulnerabilities	75
6.2.2	Description of the simulation application	76
6.2.3	Testing scenarios and expected result	78
6.2.4	Testing specification	79
6.2.5	Results of the evaluation	80
7	Conclusion	84

Abstract

Quality assurance is an important part of the software development process. Defects found in a software can lead to high economic damage for companies and their customers. [Dic11] One part of the software quality assurance is the security testing process. The goal is to find as many security vulnerabilities as possible, because hidden security errors can be found and exploited by an attacker. One way to do so are automatic security testing tools. Nowadays different approaches and tools for automatic security testing exist. The main problem is that they are not capable of revealing complex attacks like for example SQL injections or XSS attacks. A possible solution for this problem is the usage of an automatic security testing tool that sends malicious input values to the system under test and tries to detect, if a security vulnerability can be revealed. The goal for this thesis is to use retesting in order to improve such an automatic security testing tool in its ability to find hidden vulnerabilities like SQL injections.

At first a literature search is done to find such automatic security testing tools and to explore how these reveal vulnerabilities. Most of the tools found only check, if an application crashes, after malicious input was sent to the application. The crash is the only indication used to reveal vulnerabilities. Therefore vulnerabilities, where the application does not crash, like SQL injections, can not be found. A further literature search is done to retesting itself and all the different concepts that are needed for retesting in automatic security testing tools, like for example how automatic learning can be done or what indications for security errors can be found on a tested system.

Based on this knowledge a self elaborated retesting concept is introduced: In a learning phase, the automatic security testing tool learns the normal behavior of the system under test (like the normal response time or the normal CPU usage). In the following testing phase, malicious input values are sent to the system. If an abnormal behavior is found, then a retest has to be done. This is needed to reassure the result of the first test. This concept is implemented in an already existing framework of an automatic security testing tool. The results of a proof of concept show that retesting not only helps to reveal vulnerabilities but it can also reduce the number of false positives.

Kurzfassung

Qualitätssicherung ist ein wichtiger Teil des Softwareentwicklungsprozesses. Fehler in der Software können einen hohen wirtschaftlichen Schaden für Firmen und deren Kunden bedeuten. [Dic11] Daher ist die Qualität dieser Software von großer Bedeutung, die unter anderem mit Hilfe von Sicherheitstests hergestellt werden kann. Das Ziel dabei ist es, so viele Sicherheitslücken wie möglich zu finden, weil versteckte Schwachstellen von einem Angreifer gefunden und ausgenutzt werden könnten. Eine Möglichkeit hierfür ist die Verwendung von automatischen Sicherheitstestwerkzeugen. Heutzutage existieren verschiedene Ansätze und Werkzeuge, um automatisierte Sicherheitstests durchzuführen, allerdings können diese komplexere Angriffe, wie zum Beispiel SQL Injections oder XSS Attacken, nicht aufdecken. Eine mögliche Lösung für dieses Problem ist die Verwendung eines automatisierten Sicherheitstestwerkzeuges, das bösartige Eingabewerte an das zu testende System sendet und dann versucht herauszufinden, ob damit eine Sicherheitslücke gefunden werden konnte. Das Ziel dieser Arbeit ist es, ein automatisiertes Sicherheitstestwerkzeug mit Hilfe von Retesting zu verbessern, sodass es versteckte Sicherheitslücken wie zum Beispiel SQL Injections findet.

Zunächst wird eine umfassende Literaturrecherche durchgeführt, in der automatisierte Sicherheitstestwerkzeuge vergleichend dargestellt wurden. Es wird untersucht, wie Sicherheitslücken erkannt werden, und es hat sich gezeigt, dass die meisten dieser Tools nur überprüfen, ob es zu einem Absturz kommt, nachdem bösartige Eingabewerte an die zu testende Applikation gesendet wurden. Dieser Absturz wird als einziges Indiz für die Entscheidung herangezogen, ob eine Schwachstelle gefunden wurde. Daher können Sicherheitslücken, bei denen es zu keinem Absturz kommt, wie zum Beispiel SQL Injections, nicht gefunden werden. Weitere Literaturrecherchen werden durchgeführt zu dem Thema des Retestings selbst sowie zu verschiedenen Konzepten, die notwendig sind, um Retesting im Rahmen eines automatisierten Sicherheitstestwerkzeuges umzusetzen. Beispiele hierfür wären automatisiertes Lernen oder welche Indizien für Sicherheitsfehler auf einem getesteten System gefunden werden können.

Basierend auf diesen Recherchen wird in dieser Arbeit ein selbst erarbeitetes Retesting Konzept vorgestellt: Zunächst erfasst das automatische Sicherheitstestwerkzeug im Rahmen einer Lernphase, wie das normale Verhalten des getesteten Systems wie zum Beispiel die normale Antwortzeit oder die normale CPU Auslastung ist. In der darauf-

folgenden Testphase werden bösartige Eingabewerte zu dem System gesendet. Wenn dabei ein abnormales Verhalten beobachtet wird, wird ein Retest durchgeführt. Das ist notwendig, um das Ergebnis des ersten Tests zu bestätigen. Dieses Konzept wird in ein schon bestehendes Sicherheitstestwerkzeug eingebaut. Die Ergebnisse im Rahmen eines Machbarkeitsnachweises zeigen, dass Retesting nicht nur hilft, Sicherheitslücken aufzuzeigen, sondern es kann auch die Anzahl der gemeldeten false positives reduzieren.

Chapter 1

Introduction

1.1 Problem definition

Quality assurance is an important part of the software development process. It helps to prevent errors in the software, that could lead to high economic damage for companies and their customers. [Dic11] The examination of software with the help of tests is a very important method for quality assurance however, the unintentional dysfunction of software is not the only problem, but also the deliberate manipulation and attacks by offenders, who want to cause damage. Here too, software has to be tested before using it in a live environment in order to find as many potential security leaks as possible in advance and to fix them.

According to this, testing with particular emphasis on security is a very important part of the software development process. So the danger of attackers who could find and exploit security holes in the system should be minimized. One way of simplifying the process of testing and of making it more efficient is the automation of the security tests. In doing so, the tester gets a quick overview of potential threats and then he can explore them in detail. There are already different approaches of automatic security testing, which are not suitable for finding complex attacks, where the application does not crash (like for example Cross-Site-Scripting attacks). But finding this kind of attacks is necessary to minimize the amount of security leaks in the software.

1.2 Expected results

Automation of security tests ensures a faster and cheaper testing process than using manual tests, which lead to a higher security of the application. Another advantage is that the tests can be started during the development process and not just afterwards when the implementation is already finished.

It is important that these tests can always be used, no matter which application is tested.

Goal of this work is to optimize the testing process by revealing vulnerabilities when

testing applications. This should be accomplished with the help of retesting. If a security test reveals abnormal behavior of the tested application then a reassurance is done with the help of another test. Furthermore, to distinguish between abnormal and normal behavior of a system, automatic learning has to be done, to acquire knowledge of the normal behavior. Another goal of this thesis is to implement these concepts in an already existing security testing tool prototypically.

Furthermore a proof of concept will be done by testing an application that contains a vulnerability and one that does not, to show that the concept of retesting is actually useful for revealing vulnerabilities.

Summary of contributions:

- A self-elaborated retesting concept is designed.
- This retesting concept is implemented in an already existing framework for automated security tests. [TSH⁺10] [STP⁺11]
- A proof of concept for this retesting concept is done.
- A paper about an approach to automatically detect vulnerabilities in VoIP Soft-phones is published. [STP⁺11]

1.3 Structure of this thesis

This thesis is structured as follows: In chapter 2 a brief introduction to software security in general is given. Chapter 3 describes the software testing process and which methods and types of software tests exist. Furthermore it is explained how security testing and the automation of security testing can be done. In chapter 4 a specific framework for automatic testing is discussed. This framework is the basis for this thesis and concepts of how to improve it so that retesting can be done is discussed in this section. In chapter 5 the elaborated retesting concept itself is presented that helps to accomplish the goal of finding security errors. In chapter 6 it is described how these concepts were implemented in the fuzzing framework. Afterwards an evaluation is done as a proof of concept that retesting helps to reveal security vulnerabilities. In the end a conclusion is given in chapter 7.

Chapter 2

Introduction to software security

In this chapter basic knowledge concerning software security and also the importance of software security are explained.

2.1 Importance and problems of software security

The number of computer services available nowadays is growing and therefore also the relevance of computer security increases. [HS12] Most attacks nowadays aim at the software itself. The security awareness in public and in business rises and so does the request for secure software. [NBA10]

The problem with software security is that it is nothing that can be added to a software in the end, if it is not already considered during the design and implementation phase of the software development. It already has to be considered in the design phase of the software development, where the architecture is built. [SM11] The knowledge gained during requirements and design phase can be used during the testing phase to improve the test cases. [McG06]

Software security should also not be mistaken for the implementation of security features, like the encryption of data. [NBA10]

According to Shirvani et al., Security has to be a "structured process". [SM11] In [HS12], Hazeyama et al. claim: "Software security deals with security during the whole software development process." [HS12]

This can be achieved with the help of the security development lifecycle. One part of this lifecycle is the testing of the implemented software which can also be done automatically.

2.2 Terminology concerning attacks

Terms that will be used throughout this thesis are explained in advance in this section.

2.2.1 Bug

A bug is an error that exists in the source code of the software and can be revealed when the source code is executed. [McG06] Some bugs are easily found (especially if they are in a part of the software that is often used), but others can stay hidden for a long time (months or years). [TLJ⁺12]

2.2.2 Flaw

A flaw is a problem that lies in the architecture of the software. [McG06] An example is when sensitive data (like passwords or health data) is sent over the network without encryption.

2.2.3 Vulnerability

A vulnerability is a special kind of bug. An attacker could use it to provoke an unintended behavior of the software. If furthermore the input values or the steps are given that are needed to use the vulnerability, then this is called an *exploit*. [BNS⁺08] One type of vulnerability is the "zero-day vulnerability" that is neither patched, nor known to the public. [SHE12b] One possibility of how to reveal these kinds of vulnerabilities is with the help of automated test procedures. [BNS⁺08]

2.3 Properties of secure software

In literature there are often used following two models to describe the properties of secure software: the CIA-triangle and the Parkerian Hexad. [HHSB10]

2.3.1 CIA-triangle

The three main properties of secure software are: [And11]

- Confidentiality
- Integrity
- Availability

These three together are called the CIA-triangle or the CIA-triad and can be seen in figure 2.1.

Confidentiality

Confidentiality means that the information is only readable to users who are authorized to do so. It refers to the ability of the data-owner to protect the data. Confidentiality can be compromised for example if the password and the username of an account get stolen or information is sent to a wrong person by mistake. Confidentiality is important



Figure 2.1: CIA-triangle [And11]

for all kinds of data that has to be protected against unauthorized view - like for example private emails, health data or secret information of governments or companies. [HHSB10] [And11]

Integrity

Bishop state that "Integrity refers to the trustworthiness of data or resources". [Bis03] It can either be guaranteed the data can not be changed by anyone who is not authorized to do so or unauthorized changes can be detected. An example for providing integrity is permission systems in operating systems, where the owner of a file can decide who else is allowed to edit or delete his file. An example, where integrity is important, is the information in a patient's file in a hospital. If this file contains wrong (because it was changed by someone unauthorized) data, then this could lead to wrong decisions concerning the treatment, which again could lead to undesirable results for the patient. [HHSB10] [And11] [Bis03]

Availability

Availability means that the data is accessible whenever it is needed. An example of compromised availability is, when the server holding the information is not reachable. If the availability is affected by an attacker, this is usually called a Denial of Service (DOS) attack. [HHSB10] [And11]

2.3.2 Parkerian Hexad

A variation of the CIA-triangle is the Parkerian Hexad introduced by Donn B. Parker. It consists of confidentiality, integrity and availability like the CIA-triangle, but adds: [HHSB10]

- Possession/control

- Authenticity
- Utility

Possession/Control

Possession describes, who the owner of the data is. The difference to confidentiality is: An unauthorized person got hold of the data (is in possession of the data), but the data is not readable because it is for example encrypted - so the confidentiality is not violated. [Gol12]

Authenticity

Authenticity tells, who the author of the data was. Concerning electronic information this could be achieved with the help of digital signatures. [HHSB10]

Utility

Utility tells, how useful data is. If for example important information gets encrypted and stored on a disk, but the key for decryption is lost, then the data is still available, but not in a useful way, because it can not be decrypted. [HHSB10]

Utility should not be confused with availability. If there is a utility problem, then the data is still available - it is just in a wrong format. [HHSB10]

Chapter 3

Software security testing

In this chapter security testing is described. Therefore first an overview of software testing in general is given including the whole software testing process. Based on this knowledge, security testing, which is a special case of software testing, and the possibility to automate security tests are discussed.

3.1 Introduction to software testing

Myers et al. stated that "Testing is the process of executing a program with the intent of finding errors". [Mye11]

The main goal of software testing is to raise the quality of the software. [OI10] This is achieved by finding as many defects as possible. The more defects are found, the less will be in the software. [BJC10] Testing is a process, means that testing only once during the software development process is not enough. [GVVEB08]

Software testing can be categorized by different aspects. One criteria is the method that is used when testing the software: there are black box tests, white box tests and gray box tests. Another criteria is which aspects e.g. the functionality, performance or security are tested.

3.1.1 Software testing methods

Based on the knowledge which the tester has while performing the tests, there are three different methods of tests: black box tests, white box tests and gray box tests.

Black box tests

When performing a black box test, the tester has no knowledge of the internal structure of the software. The software behaves like a black box - the tester only knows the

interfaces and the expected input parameters, but not the source code or the software architecture. So he can only send different values to the system under test (SUT) and check if the system responds in an expected manner. [MVS10] [OI10]

White box tests

The opposite of a black box test is a white box test, where the tester has full access to all internal structures of the software. This includes the source code and the software architecture. This knowledge can be used to create test cases that check different code paths. Especially boundary conditions can be tested better this way than with the help of black box tests. [MVS10][OI10]

Gray box tests

Gray box testing is a mixture of black box testing and white box testing: The tester knows parts of the internal structure. So the work can be done in a more efficient way, than with black box testing. [BD09]

3.1.2 Types of software tests

Software tests can be divided in two groups: functional and non-functional tests that test the functional and the non-functional requirements. Security requirements can also be either functional or non-functional and therefore can also be tested with functional or non-functional tests.

Functional tests

The purpose of functional tests is to check, whether external (functional) requirements are met. [Lig09] Functional requirements define, which functionality the program should have, how the program reacts in different situations with different inputs and what the expected output should be. [Wan04]

A black box approach is often applied for functional tests, because all test cases have to be generated based on the specification and so the internal structure does not matter. On the other hand, not all black box tests are functional tests - for example the so called random testing or fuzzing where random generated values are used as input for the software. [Lig09]

The problem with functional tests is that it can not be guaranteed that all paths in the source code were tested. [Lig09]

Non-functional tests

Non-functional requirements describe quality attributes that have to be met by the software, for example performance, maintainability or security. Non-functional tests are used to test these requirements. [AACF12]

This thesis covers security tests as a specific kind of non-functional tests.

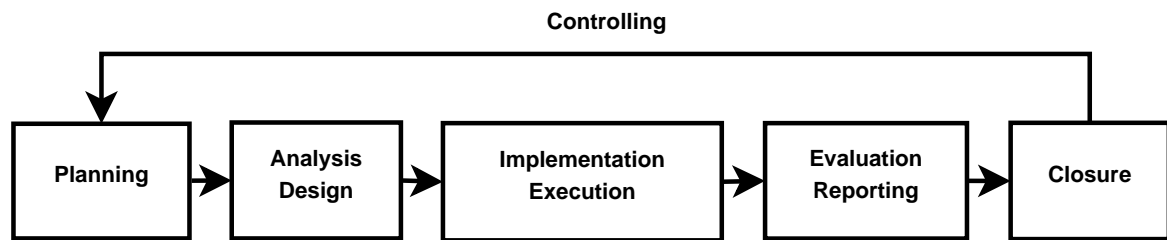


Figure 3.1: Steps of a common testing process

3.2 Software testing process

The software testing process consists of different steps: [GVVEB08]

- Planning and control
- Analysis and design
- Implementation and execution
- Evaluating exit criteria and reporting
- Test closure activities

These steps can be seen in figure 3.1 and are explained in detail in the following subsections.

3.2.1 Planning and control

At first the so called mission of the testing has to be defined: it consists of the goals that all stakeholders of the project have and all the risks the software is facing. Based on this knowledge, the following has to be declared based on the book of Graham et al. [GVVEB08]:

- **What are the goals of testing?** - Based on that, the approach and the testing plan can be chosen.
- **What is the scope of testing?** - The scope can be a software, a system or a component.
- **What is the purpose of testing?** - Is the testing done to find defects or to show that it fulfills the given requirements?
- **What is the test approach?** - Which testing techniques are used? What are the critical parts of the system and what coverage is necessary? Who is involved in the testing - the developers, the users? Is there any special software that will be used for the testing?

- **What is the test strategy?** - If a document exists in the company that defines test strategies, then the testing has to match these standards. If there are any reasons not to follow the standards, then this matter has to be discussed with the stakeholders to find a solution for this problem. The test strategy is the testing approach on a high level (e.g. all the tests should be automatized, so that they can be re-run).
- **Which resources are needed for testing?** - This is the detailed planning of which persons are involved in planning and how the test environment looks like.
- **What is the time schedule for further steps?** - The time scale for the different activities in the design-, the implementation-, execution- and evaluation-phase of the testing process have to be defined so that they run smoothly.
- **What are the exit criteria?** - The exit criteria are needed to define, how much testing is enough, ergo when the testing process is finished. An example for exit criterion is, how high the code coverage has to be.

Besides planning the testing activities, also the act of monitoring and controlling is important. During every step of the ongoing software testing process, it has to be checked if the current state of the process matches the planned state. Then the results are reported to the stakeholders. If planned and actual situation do not match, then actions have to be taken - for example the plan has to be adapted.

The purpose of test control is to measure and monitor the testing itself and the progress in order to provide data for stakeholders via reports and to provide the people responsible for planning with further information so that decisions for the testing process can be made. [GVVEB08]

3.2.2 Analysis and design

The information gathered in the planning phase is used in the next step for analysis and design. At first all documents available for the project (the so called test basis) like requirements and architecture are reviewed. Based on this information the tester is able to find errors in the specification and to help prevent defects at an early stage of software development. The goal of the tester in this phase is to define test conditions. These are components of a system that can be tested with the help of test cases like for example input fields. Test cases are built during the implementation phase for all parts of the software with special attention to the risks found in the planning phase. To accomplish this goal it also has to be checked, if the specifications are testable. [GVVEB08]

3.2.3 Implementation and execution

The goal of the implementation and execution phase is to create test cases and to set up a testing environment where the tests can be run. The activities during the implementation and execution phases(based on the book of Graham et al. [GVVEB08]) are:

- **Implementation**
 - **Creation of test cases** - At first concrete test cases are defined including suitable test data. Furthermore, so called testing procedures have to be defined that tell how the test cases should be done. If scripts for automated testing are needed or wanted, they also have to be developed in this phase of the testing process.
 - **Definition of test suites** - Logically related test cases are combined to test suites, so that testing can be done in a more efficient way.
 - **Implementation and checking of the test environment** - The environment for testing has to be set up and it has to be examined if it is ready for testing.
- **Execution**
 - **Execution** - At first the test cases or test suites are executed as defined in the testing procedures before.
 - **Documentation** - For every test case it has to be documented in the testlog, what was exactly tested and which version of the software was tested.
 - **Checking the results** - The next step is to find out for each test case, if the actual behavior of the software during the tests corresponds with the behavior that was expected. If there is a difference, then the reason for this difference has to be found: Is there an error in the software under test, is the test data or the test case wrong or is there a flaw in the way the testing is done? The errors have to be reported.
 - **Retesting** - If the tester gets the information that the found error was fixed, the test cases have to be repeated to reassure that the defect was corrected. It is not enough to re-test only the test case that found the error, because the fixing of the defect could have caused side effects in other areas of software, which could lead to other errors.

3.2.4 Evaluating exit criteria and reporting

In this phase, it has to be checked, if the exit criteria defined in the planning phase before are met with all the test cases that have been done so far. If this is not the case, further testing has to be done. Otherwise the execution of test cases can be stopped and a report of the testing activities has to be written, so that the stakeholders can be informed of the testing state of the software. [GVVEB08]

3.2.5 Test closure activities

The last step in the testing process are the test closure activities. It will be checked, whether all defects found in the project are resolved and all scripts and data used are archived. This is necessary in order to use them in future testing for the current or other

projects. The information gathered is used as a lessons-learned material to improve the testing process itself. [GVVEB08]

3.3 Security testing

The purpose of security testing is to ensure the confidentiality, integrity and availability of a system. [WM10] [RMZR10] Ideally, after security testing, there should be no more vulnerabilities in the software that an attacker could exploit - then the software can be called secure. [Wan04]

One problem of security testing is that nothing is one hundred percent secure - if no more errors can be revealed, this does not automatically indicate that no further vulnerabilities exist, just that all the test conditions used so far, could not detect a security breach. [McG06]

Nunes et al. state that a problem is that "[...] software engineers think that an identify and authentication control implemented in software to protect data confidentiality and integrity makes this software secure". [NBA10] These security features can be specified with the help of requirements and can be tested by using functional tests. The problem with security is that it can not be achieved by only implementing security functionalities like for example the encryption of the network traffic and testing if these functionalities work as specified. [McG06]

Security is the absence of unintentional software behavior that can be exploited. [McG06] Unlike testing if functionality is implemented as expected, security testing can be seen as checking that unwanted behavior is not existing. Therefore it is also called "testing for the negatives". [McG06]

Like already explained before in section 2.1, security is not a feature that can be added at the end of the software development, but has to be considered throughout the whole development process. This can be achieved with the help of the security development lifecycle that is explained in the following section.

3.3.1 Security development lifecycle

One famous quote of Bruce Schneier is: "Security is a process". [Sch00]

Security has to be kept in mind at every step of the software development lifecycle. Otherwise it is very hard or even impossible to create the same level of security as if it had been considered right from the start. [RMZR10]

Different kinds of security development lifecycle methodologies exist: [Gee10]

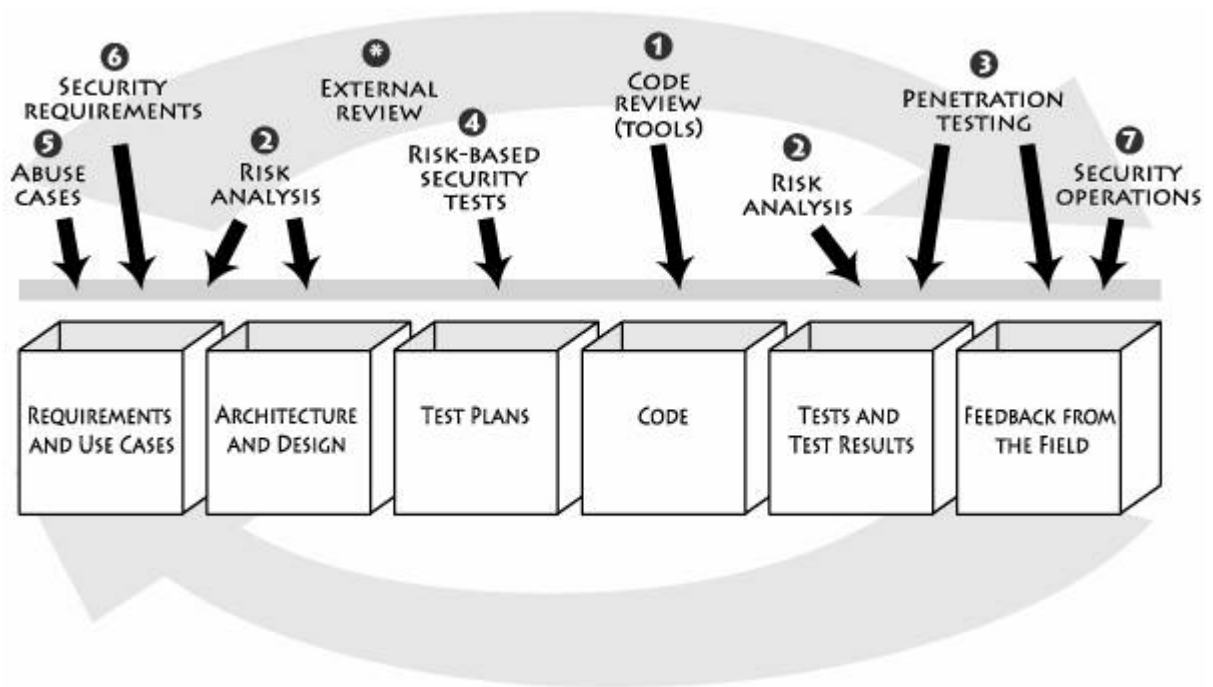


Figure 3.2: One example for the security development lifecycle: 7 touchpoints [McG06]

- *Comprehensive, Lightweight Application Security Process* (CLASP) [GBDW⁺07] [NBA10]
- *Microsoft's Security Development Lifecycle* (SDL) [GBDW⁺07] [Gee10]
- *Software Assurance Maturity Model* (SAMM) [Gee10]
- *Building Security in Maturity Model* (BSIMM) [Gee10]
- *Secure Software Development Lifecycle* (SSDL) [Gee10]
- *7 Touchpoints* [McG06]

All of these models have the goal that security is already considered during the software development. [Gee10] As an example, one of these models is explained in detail in the following.

The security development lifecycle model defined by McGraw, called the 7 touchpoints, can be seen in figure 3.2. At every step of the software development process, security is kept in mind. In the end the feedback from the software operation is taken

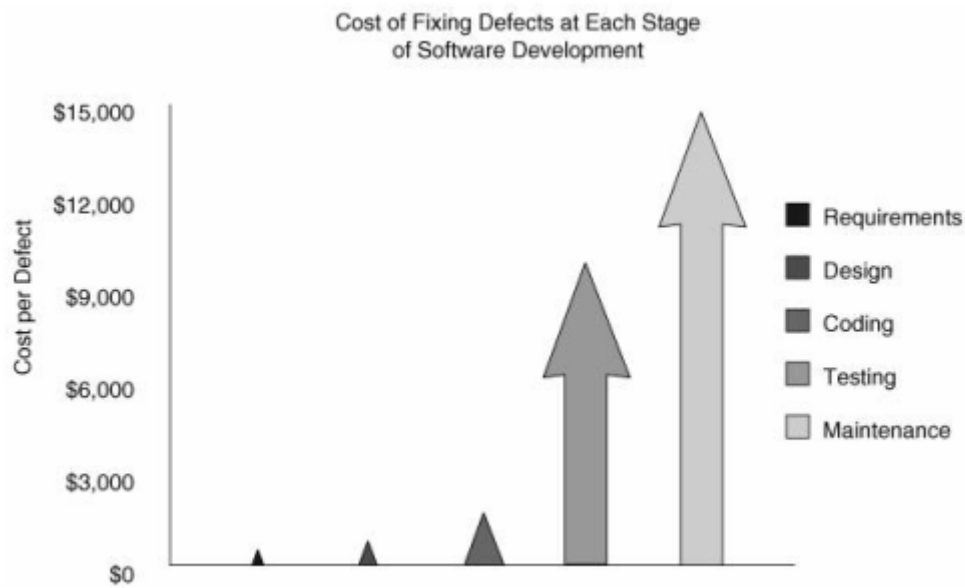


Figure 3.3: The costs to fix an error in each phase of the software development lifecycle. [McG06]

and considered further in the next planning phase so that the lifecycle is closed and can be improved. [McG06]

Furthermore the earlier an error is found, the cheaper it is to fix it. [BP88] This can also be seen in figure 3.3: If an error is found in the requirements or the design phase, fixing it is very cheap. But if the same error is discovered and fixed in the testing or maintenance phase it costs many times more.

The kind of software development process seen in figure 3.2, is only an example - it is also possible to adapt other software development processes, like the V-model or agile models (for further information also see [GVVEB08]), by adding security measures like code review or risk analysis to the processes. [McG06]

All the security measures are described in detail in the following sections.

3.3.2 Abuse cases

The creation of abuse cases is part of the requirements and use cases phase of the security development lifecycle as can be seen in figure 3.2.

Abuse cases describe the expected behavior of the system when it is under attack. It has to be described, who a possible attacker could be, what the target of the attacker is and how long the target should be protected. [McG06]

3.3.3 Security requirements

The goal of the security requirements phase is to create functional security requirements. An example would be the encryption of the network traffic. Furthermore, on basis of the abuse cases, described before, requirements should be created to prevent the attacks discussed in the abuse cases. [McG06]

3.3.4 Architectural risk analysis

As can be seen in figure 3.2 the risk analysis has to be done at different stages in the security development lifecycle:

- Requirements- and use cases-phase
- Architecture- and design-phase
- Testing-phase

The goal of architectural risk analysis is to find flaws in the design of the software. This is done by the examination of the architecture and the design on basis of possible attack scenarios. [McG06] Examples for risks are: Sensitive data is sent over the network without encryption. Passwords are stored in the database in plaintext.

3.3.5 Risk-based security tests

As part of the test plan (as can be seen in figure 3.2) risk-based security tests have to be created as well.

There are two kinds of tests that have to be done to test the security: [McG06]

- **Testing the security features** - An example would be to test if the encryption of the network traffic works as intended. This can be accomplished with tests that check, if the functionality works as described in the requirements.
- **Testing if unwanted behavior exists** - These tests are created on basis of "attack patterns, risk analysis results and abuse cases"[McG06] that were created in the steps before. They should ensure that this bad and unwanted behavior can not be accomplished in the software by an attacker.

3.3.6 Code reviews

Code review can be either done manually or automatically with the help of static analysis tools. [McG06] Static analysis tools scan the source code, looking for special structures or words that could indicate a bug without having to run the program. [Wan04] For example: If the source code is written in C and the function "gets" was used by the programmer without checking the length of the buffer and the input value, then the risk

of a bufferoverflow is given at that position. The bugs found by the tools still have to be checked manually to find out if a bug was found or if the tool just reported a false positive, which means that the found bug is no bug after all.

One possible advantage of the usage of tools is that testing can be quicker than manual testing. Therefore it can be done more often during the development lifecycle and so more bugs can be found. Ideally the testing with the static analysis tool can be done during the programming phase by the developers themselves, but they have to be trained before, otherwise they would not be capable of interpreting the results of the tools and fixing the problems. Furthermore, if the tool produces too many false positives and the developers spend more time interpreting the result of the tools than on fixing security bugs, the developers will refuse to use it, which results in fewer security bugs being found. [McG06]

The problem with code reviews is that only bugs can be found, but not the flaws in the architecture that account for about 50 percent of all security problems in a software. This is the reason, why architectural risk analysis is as important as code review. [McG06]

3.3.7 Penetration testing

The purpose of penetration testing is to reveal as many security bugs in the system under test (SUT) as possible. However, if no further bugs can be found, this does not mean that the software is free of bugs. To improve the effectiveness of penetration testing, it is important that is done in an organized way: All the outputs of the phases of the security development lifecycle before the testing phase have to be considered - especially the risk analysis and the abuse cases.

In the phase of penetration testing it is also important to test the software in its final environment, check if the configuration is done correctly and if the security features are implemented and work as intended. [McG06] In the phase of penetration testing, automated security testing like for example static analysis or fuzzing can be used.

3.3.8 Security operations

The last step of the security development lifecycle is to gather information during the operation of the software. Part of this phase is to configure the software and to consider all the environment-layers the software depends on like the network-layer and the operating system. Furthermore the logging and monitoring of the system is important so that in case of an incident, the attack can be noticed and information can be gathered. This information can also be used to improve the software development lifecycle itself. [McG06]

3.4 Test automation to reveal security vulnerabilities

There are different approaches to automatically test a software with the goal of finding security bugs like for example:

- Static code analysis
- Fuzzing

The topic of fuzzing is discussed in detail in this section including the common architecture of fuzzers. It is the basis for this thesis, because most of the time it helps to reveal software vulnerabilities in a cheaper and more effective way than static code analysis. [LJC09] It is located in the penetration testing phase of the SDL and it uses the knowledge gained in the phases before. [McG06]

3.4.1 Introduction to test automation in software testing

The main purpose of test automation is to make testing faster and less expensive than manual testing - for details see [BCR08]. There are different types of tests that would be very hard or impossible to perform without automation. [XJ10]

Two examples are given: [KBP02]

- **Stress tests** - The goal of stress tests is to observe how the system behaves under heavy load [GVVEB08] and what the upper limit of requests is. [ANNM06] For example: A telephone system should be tested to verify, how many calls can be handled in parallel, before the system does not respond any more. A manual test of this scenario is very complicated, because many telephones and people would be needed, who call the telephone system at the same time. These kinds of tests are also important for security, because they simulate a Denial of Service (DOS) attack. [ANNM06]
- **Race conditions** - Race conditions are problems that need special timing, so that they can be observed. This is also the reason, why they are very hard to detect. [KBP02] They typically occur in "parallel and distributed systems". [JSF06] Automatic testing can help to reproduce these errors by varying the timing of the taken steps in the test. [KBP02]

3.4.2 Advantages of test automation for revealing vulnerabilities

The automation of tests has several advantages in comparison to manual testing for functional tests. Some of these advantages can also be applied to the test automation of security testing: [KBP02]

- **Cost reduction of testing** - At first effort and money have to be invested to automate the test. Every further test is cheaper (because it is faster than manual testing) and it can be run by anyone. [KBP02] This can also be applied to automatic security tests that can be carried out by the developers during the development phase. [McG06]
- **Provision of early feedback** - The advantage of early feedback is (as can be seen in figure 3.3), is that the earlier a bug is found, the cheaper it is to fix it. Therefore it is better to fix the bug in the development phase than in the testing or maintenance phase afterwards. These bugs can be found by the developers with the help of testing tools where they can for example perform a static code analysis. [McG06]
- **Reduction of risks** - The test can be run automatically at every build. For every new version of the software an automatic security test can be done. This helps to check if new vulnerabilities were introduced into the system. [KBP02] [Gli93]
- **More efficient testing** - When trying to find vulnerabilities, all the interfaces of the software have to be tested against a variety of malicious input values. With test automation this can be achieved in a more efficient way than with manual security testing. [99]

3.4.3 Static code analysis

Static code analysis checks the source code of a software to find patterns that indicate a vulnerability. This can be done automatically and the source code does not have to be executed. Suspicious structures are reported to the tester. [PAR⁺12] Static code analysis is used to find vulnerabilities in an early stage of the software implementation phase. [Bac10]

3.4.4 Introduction to fuzzing

Fuzzing is mostly seen as a black box or gray box testing method [TDM08], but also white box fuzzing exists [GLR09].

The idea of fuzzing is to send malicious or unexpected inputs to the system under test (SUT), to verify the behavior of the system and to check if a security vulnerability can be revealed. This input can be created in multiple ways, for example it can be created randomly or with the help of so called attack vectors, which are a list of known, potentially bad input values. [TDM08] Furthermore the input can also be generated by the fuzzer itself according to a specification (generation-based fuzzing) or known, valid data are mutated randomly. [AFWJJMZ11]

To verify if the attack done by the fuzzer was successful, it is checked if the system crashes or any other indication for an occurred vulnerability can be found. Therefore, fuzzing is a possible method to find zero-day attacks, which are attacks that are totally new for the SUT. Fuzzing is able to cover more test cases in shorter time, than could ever be done manually. [TDM08]

The advantage of fuzzing to static code analysis is that normally it is cheaper and more effective. [LJC09]

Fuzzing is, as it is typical for security tests, a testing for the negatives. [TDM08] [TDB12] This means that fuzzing does not check, if specified features work as expected, but that no unexpected or unwanted behavior occurs. [McG06] Fuzzing is also one field of application of data-driven test automation.

Data-driven test automation

Data-driven test automation means that different input values, taken from a data source, are used for a test case. For example: A search-function should be tested with multiple different input values. Therefore a file is created where in every line a new input value and the expected result (the number of findings of the search function) is put. When the automatic testing script is started for this test case, it takes one line of the input file after the other, takes the input values, runs the search function with these input values and afterwards compares the result of the search function with the expected result. In the end all differences to the expected result found are reported. [KBP02]

The advantage of data-driven test automation is that new tests can be created very easily: the test script has to be written only once and whoever creates the actual tests, does not have to have any programming skills - he or she only has to add new values to the data source. [KBP02]

3.4.5 State of the Art of fuzzing

In the literature there are fuzzers for different purposes (like for example for VoIP telephony) that are specialized for these domains. Most of these fuzzers use very little evidence to find out, if an error has occurred or not. In [HYDD08] a fuzzer is described that recognizes an error, if the application has crashed. There six different kinds of crashes are distinguished, including the case of a restart of the application or if the system is not available at all. Furthermore only three of the found six kinds of crashes are discussed, because these are the ones that can easily be recognized with the help of a debugger. The fuzzer called PROTOS works in a similar way and distinguishes between four kinds of crashes that are identified as found errors. [AERI10]

In the paper [Mul09] of Mulliner et al. about the testing of telephones, it is only checked manually whether there has been a crash or a reset of the telephone under test. Other fuzzers use a debugger, so all exceptions are written into a logfile, which has to

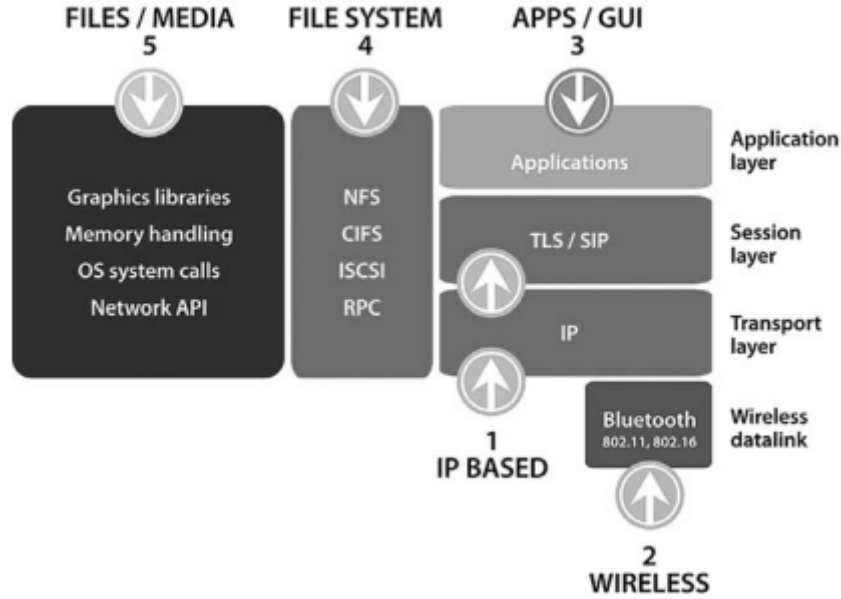


Figure 3.4: Example of interfaces where fuzzing can be useful [TDM08]

be checked manually again to decide whether an error has occurred or not. [JME08] [KHK10]

Some fuzzers use a mixture of static and dynamic code analysis for error detection. The advantage is a very efficient way of fuzzing with a low amount of false positives. On the other hand its disadvantage is that this kind of testing is not possible in black box tests where the source code is not available. [TCZ⁺09] [LWT⁺08] [ZHYS11]

The fuzzing framework, which should be extended, is already described in a paper [TSH⁺10]. Different so called Analyzers exist to monitor the state of the currently tested system. In [STP⁺11] Schanes et al. used the fuzzing framework for revealing security vulnerabilities when testing VoIP applications.

3.4.6 Area of operation for fuzzing

Fuzzing can be useful for finding bugs in different kinds of systems. Every system and every part of a system can be tested, as long as an external interface exists, where input values can be sent to. This can be any kind of application, any protocol or any other kind of software as can be seen in figure 3.4. [TDM08]

3.4.7 Kinds of fuzzers

A classification can be done based on the ability of the fuzzer to handle states: [TDB12] [TDM08]

- **Stateful fuzzers** - Stateful fuzzers are needed to test complex scenarios that do not simply consist of a request and response message. An example would be that first a login has to be done, before further steps can be taken. So stateful fuzzers have to use valid input values to accomplish the necessary steps (like the login in the example) until the actual functionality that should be fuzzed, can be tested. [TDB12]
- **Stateless fuzzers** - In contrast to stateful fuzzers, stateless fuzzers do not have to manage different states. They can only send individual requests and receive the corresponding response. [TDM08]

3.4.8 Architecture of a fuzzer

The architecture of a common fuzzer implementation is based on at least following functionalities: [TDM08]

- Protocol modeler
- Anomaly library
- Attack simulation engine
- Runtime analysis engine
- Reporting

The interaction of the different parts can be seen in figure 3.5. The parts are also described in the following subsections.

Protocol modeler

The protocol modeler is used to specify the input that is needed by the SUT. For example, if the SIP protocol should be tested, then the structure of a SIP message is needed by the fuzzer to create the right input messages for sending. This could be done for example with the help of an XML-file where the message template with one or more placeholders for the dynamic input (attack values created by the fuzzer) is defined. [TDM08]

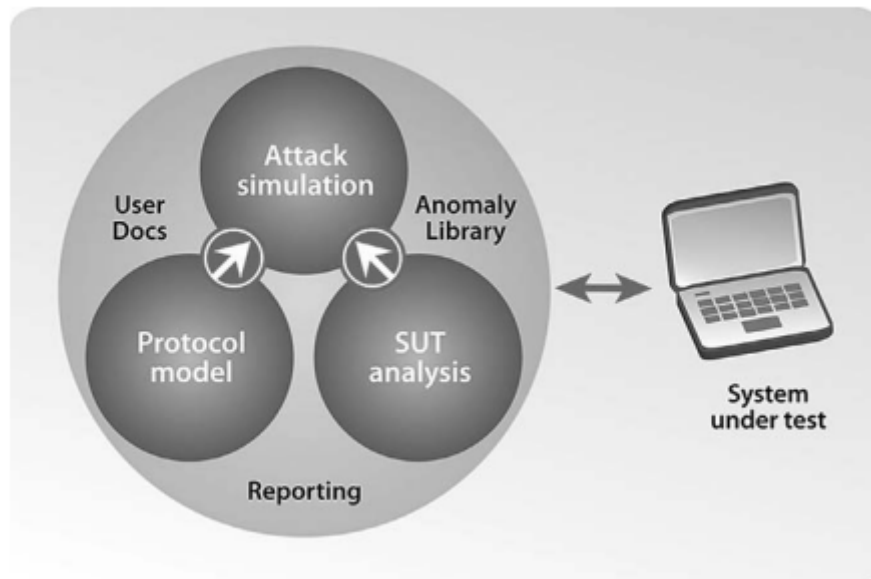


Figure 3.5: Parts of a fuzzer [TDM08]

Anomaly library

The anomaly library is needed to create values that are used as malicious input during the simulated attack. In the easiest case, these values are created randomly by the library. A more sophisticated way (that is used by most modern fuzzers) is to use values that are known to be malicious and so they are stored in the library. [TDM08] [BSA⁺11]

Tsankov et al. claim that "Fuzz-testing is intrinsically incomplete". [TDB12] This means that not all possible inputs can be tested, because the number of input values is possibly infinite. Therefore it is important that values are used that are more likely to reveal vulnerabilities like testing of boundaries or typical input values that can provoke attacks like SQL injections. [TDB12]

Attack simulation engine

The attack simulation engine uses two things: the anomaly library and the protocol modeler, which has been already mentioned before. It gets the structure of the messages to be sent from the protocol modeler and replaces all the placeholders with values from the anomaly library. So the actual message is created and can be sent to the SUT. [TDM08]

The attack simulation engine also has to be able to insert valid values into the messages. An example where this ability is needed is the fuzzing of security protocols: If the message is discarded on basis of wrong checksums or hashes, then the deeper functionality of the protocol can not be tested. [BGM⁺07]

Runtime analysis engine

The runtime analysis engine (also called SUT analysis in figure 3.5) is used to observe all the processes on the SUT. This is necessary to detect abnormal behavior that was triggered by the fuzzing - this is the way to find out, if the current test was successful and a security error could be found. [TDM08]

Reporting

The reporting is needed to inform the tester about the final outcome of the test run. Based on these results, the tester has to verify, if all found errors are actual vulnerabilities. These results are further reported to the development team, so that the found bugs can be fixed. [TDM08]

As can be seen in figure 3.5, the anomaly library and the reporting are resources that are either used or created during the fuzzing process whereas the protocol modeler, the SUT analysis and the attack simulation are part of the fuzzer program itself.

3.4.9 Description of steps for testing with a fuzzer

During a test run, multiple messages (that were created by the attack simulation engine) are sent from the fuzzer to the SUT. These messages can be either valid requests or anomalous requests and the response by the SUT is either: [TDM08]

- **Valid response** - the SUT behaves as expected. [TDM08]
- **Error response** - the SUT also behaves as expected. It detects that an anomalous request has been sent and so it responds with a error response that was specified in the requirements of the SUT. [TDM08]
- **Anomalous response** - the SUT did not behave in an expected or specified way. [TDM08]
- **Crash** - the SUT did not behave in an expected way, but crashes and no response is sent to the fuzzer. [TDM08]

An example can be seen in figure 3.6. At first a valid request and then three anomalous requests are sent to the SUT - in each of the cases the SUT responds in another way. This response then can be interpreted by the fuzzer. In case of a crash it is always certain that a security error has been found, and so it can never be a false positive. A crash can be compared to a DOS, because after the crash the SUT is not available any more. [TDM08] The anomalous response is also interesting, but it is harder to decide, if an actual security error could be found or not.

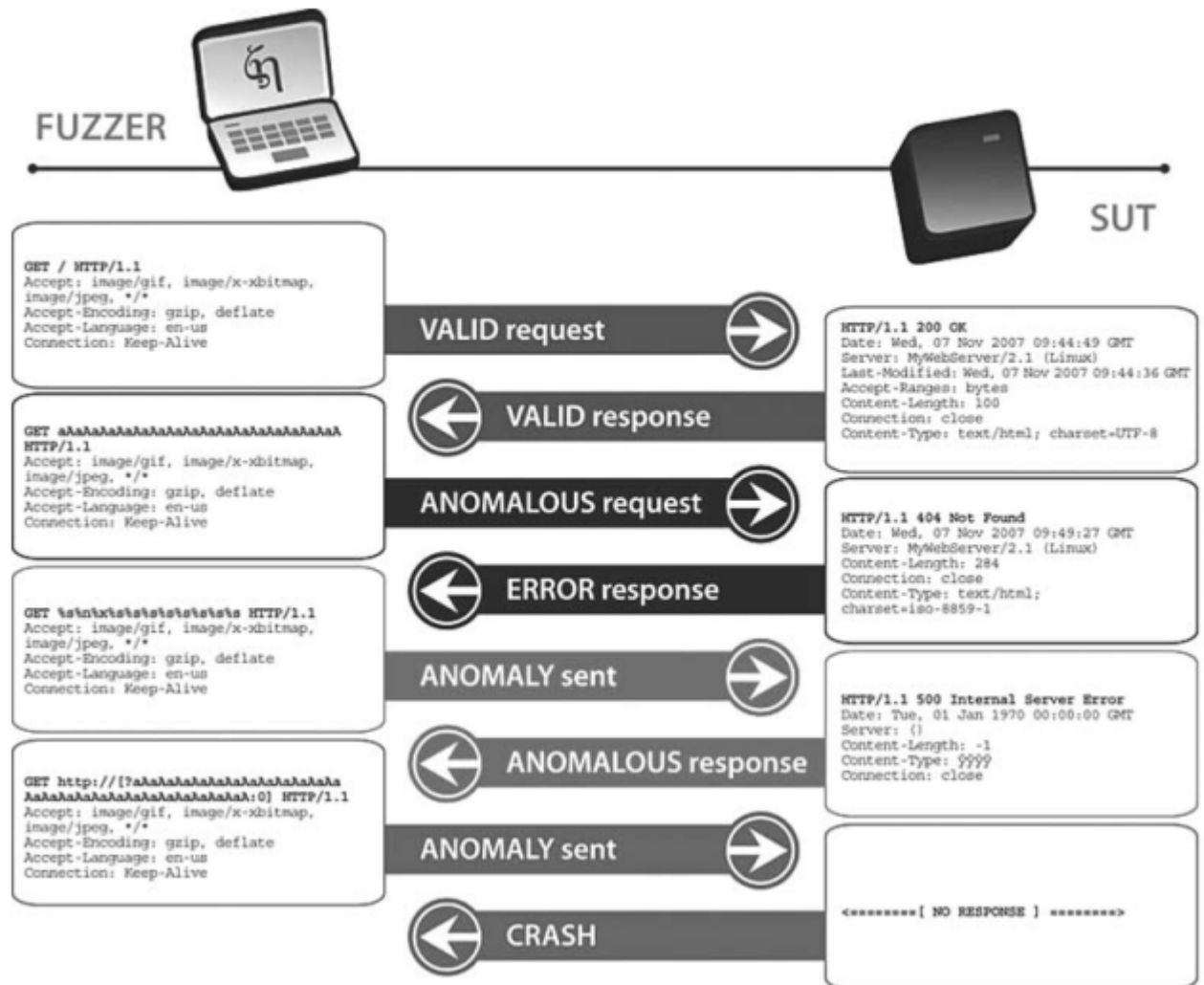


Figure 3.6: Example test run of a fuzzer [TDM08]

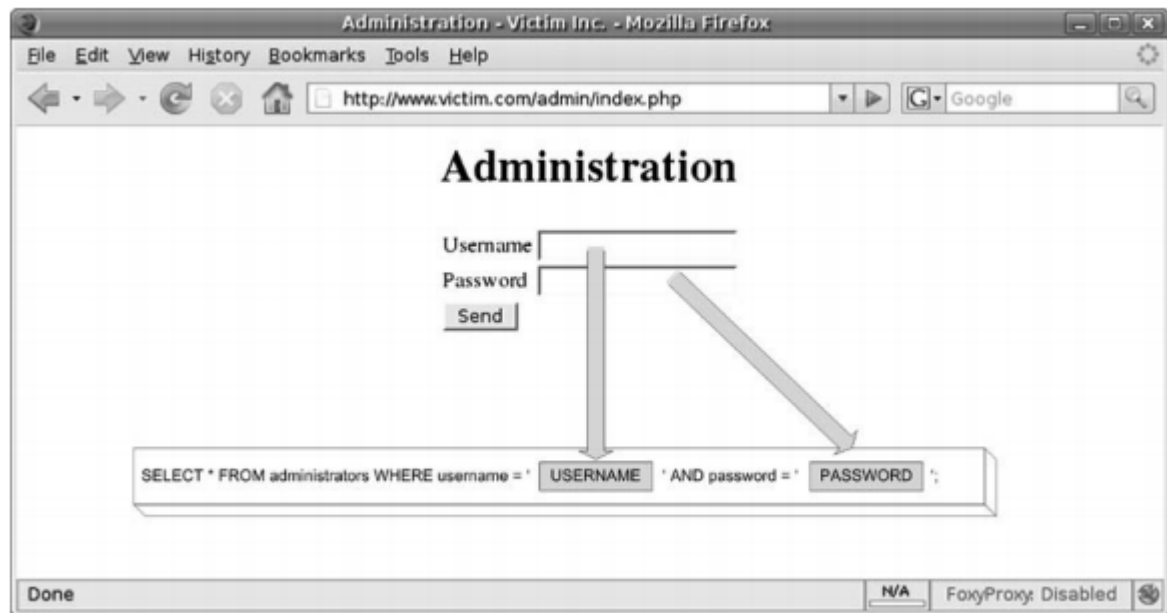


Figure 3.7: Example of a SQL injection [Cla12]

3.5 Types of software vulnerabilities

The goal of security testing is to find security vulnerabilities in the source code. In the following some kinds of vulnerabilities are explained, which are mentioned in this thesis. In [100] a list of the most frequently revealed vulnerabilities in web applications can be found.

3.5.1 Injection attacks

Injection attacks were the most frequent kind of attacks in the web in 2010 based on the information given by the Open Web Application Security Project. [100] [MSM12] When doing injection attacks, an intruder tries to "inject meta data into the process". [TDM08] Parts of SQL queries can be injected into sql queries, XML code can be injected into XML data sets, also command line injections can be done and so on. [TDM08] SQL injections are explained in the following as an example for injection attacks.

SQL injection

Nowadays most applications store their data in a database and use SQL as a query language to manipulate or receive data. If the input validation is not done correctly, an attacker is able to manipulate the query that is sent to the database. So for example, all information could be selected from the database, new data sets could be added or the content of a table could be deleted. [TDM08]

```
SELECT *  
FROM administrators  
WHERE username = '' AND password = '' OR '1'='1';
```

Figure 3.8: SQL query after a successful SQL injection [Cla12]

An example is given in the following:

A webapplication has fields for username and password, so that a user can log in (as can be seen in figure 3.7). When the "Send" button is clicked, then the values from the fields are taken and used as input value for the SQL query in the source code (as can also be seen in figure 3.7).

If an attacker chooses as password the value ' or '1' = '1 and no input validation is done, then this will be also used in the SQL query. What the final statement looks like can be seen in figure 3.8. This statement always returns all entries in the table *administrators*, unlike only one entry where the username and password are correct as intended by the programmer. So the SQL injection was successfully executed. [Cla12]

3.5.2 Cross-Site-Scripting (XSS)

In the area of web applications, Cross-Site-Scripting (XSS), is one of the most important attacks. [ZHYS11] The goal of XSS attacks is to inject scripts into a website so that they are executed. [Fla11]

An example for source code of a website that is vulnerable to XSS can be seen in figure 3.9. This script extracts this part of the URL that is after the "?"-sign and then uses *document.write* to add the found value to the content of the website. [Fla11]

If the URL of the website that contains the source code of figure 3.9, looks like the one in figure 3.10, only the word *David* is displayed on the website. However, if the URL looks like the one in figure 3.11, then the malicious script code that is part of the URL, is executed. This example malicious code only opens a dialog box that contains the text *David*. An attacker could also use this XSS vulnerability to execute any kind of code by storing the script code on another server and embed a link to the script in the URL. An example can be seen in figure 3.12. In this example the JavaScript code stored in the file *evil.js* at location *siteB* is loaded into the content of the website by the execution of *document.write*. [Fla11]

This method can be for example used to steal confidential information from the user like the session cookie, e-mail address or password. [SH10] [ZHYS11]

Basically there are two types of XSS attacks: [ZHYS11] [TDM08] [YD12]

- Reflected XSS

```
<script>
var name = decodeURIComponent(window.location.search.substring(1)) || "";
document.write("Hello " + name);
</script>
```

Figure 3.9: Example code that is vulnerable to XSS [Fla11]

```
http://www.example.com/greet.html?David
```

Figure 3.10: Example for a harmless URL [Fla11]

- Stored XSS

Reflected XSS

When performing a reflected XSS, the attacker has to send an URL to the victim that contains malicious code (like explained in the example before). After clicking on the URL, the victim is redirected to the website, where the source code is executed. [TDM08]

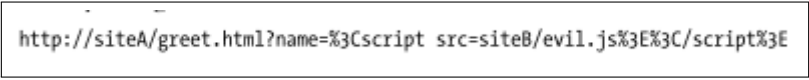
Stored XSS

In contrast to reflected XSS, stored XSS does not require action done by the user, like clicking on a URL. It is called a stored XSS, because the malicious code is already stored permanently in the database of website. The attacker can achieve this for example by writing an entry in a webforum or a guest book. As soon as a user enters the website, where the entry should be displayed, the code is loaded from the database and executed. [TDM08]

To prevent XSS attacks content filtering or input validation has to be done. [YD12] [Fla11] Encryption or the installation of a firewall do not help, because the script is triggered executed in the context of the user that has visited the website. [ZHJS11]

```
http://siteA/greet.html?name=%3Cscript src=siteB/evil.js%3E%3C/script%3E
```

Figure 3.11: Example for a URL that exploits a XSS vulnerability [Fla11]



```
http://siteA/greet.html?name=%3Cscript src=siteB/evil.js%3E%3C/script%3E
```

Figure 3.12: Example for an URL that exploits a XSS vulnerability and load external code [Fla11]

3.5.3 Denial of Service (DOS)

The goal of Denial of Service attacks is to compromise the availability of a system. [TDM08] [RRT12] The typical reason for DOS is: An attacker sends a huge amount of requests to a system. The creation of these requests and the sending does neither need a lot of time nor does it need lots of resources and can therefore be easily performed. The system that receives these request on the other hand, then has to possibly perform operations that need lots of resources. Due to this mismatch the system can not handle any more request because of its lack of resources. [TDM08]

A special kind of DOS are Distributed Denial of Service (DDOS) attacks, where the attacker uses multiple computers to send the requests to the system under attack.

There can be several reasons why the system is not available, like for example: [TDM08]

- the system has crashed
- the resources available on the system are all occupied

Some possibilities of detecting or preventing a DOS are:

If the system has crashed, this can be easily detected automatically: Regularly, requests are sent to the system and if a response is received, then the system is still alive. Furthermore, the resource usage of the system can be monitored, and if it gets too high, then the system operator is informed. [TDM08] An absolutely reliable protection against DOS does yet not exist. [XSJJ11]

3.5.4 Directory traversal

The goal of an attacker, when exploiting a directory traversal vulnerability, is to access files that he is not supposed to. If an application takes a path as an input that can be manipulated by a user, this problem can occur. [RCCP08]

Chapter 4

Architecture of the automated security testing framework

In this chapter the architecture of a specific fuzzer is described in detail. Furthermore different concepts and architectures that could help to implement retesting (as can be seen in figure 4.1) are introduced. The framework already collects some data of the SUT with the help of the Analyzers. In the next step these collected data have to be examined and indications for security errors have to be found. This knowledge can be used to automatically learn the normal behavior of the system and use it to distinguish between normal and abnormal behavior. If abnormal behavior is inspected, then a retest has to be done.

The goal of all these measures is to improve the framework in its ability to reveal vulnerability and to decrease the number of false positives reported by the framework.

4.1 Description of the automated security testing framework

The fuzzer used in this thesis has been already described in two papers about the testing of VoIP phones. [TSH⁺10] [STP⁺11]

4.1.1 Architecture of the framework

The fuzzing framework can be used for all kinds of systems like VoIP phones [TSH⁺10] [STP⁺11], web services and protocols. This is possible, because it is a very generic framework with lots of configuration possibilities. The request messages for example are generated with the help of templates, which can be easily adapted for the different applications to test.

Like shown in figure 4.2 every test setup consists of two parts: The fuzzer and a host, which is the system under test (SUT). The fuzzer itself consists of the following main parts:

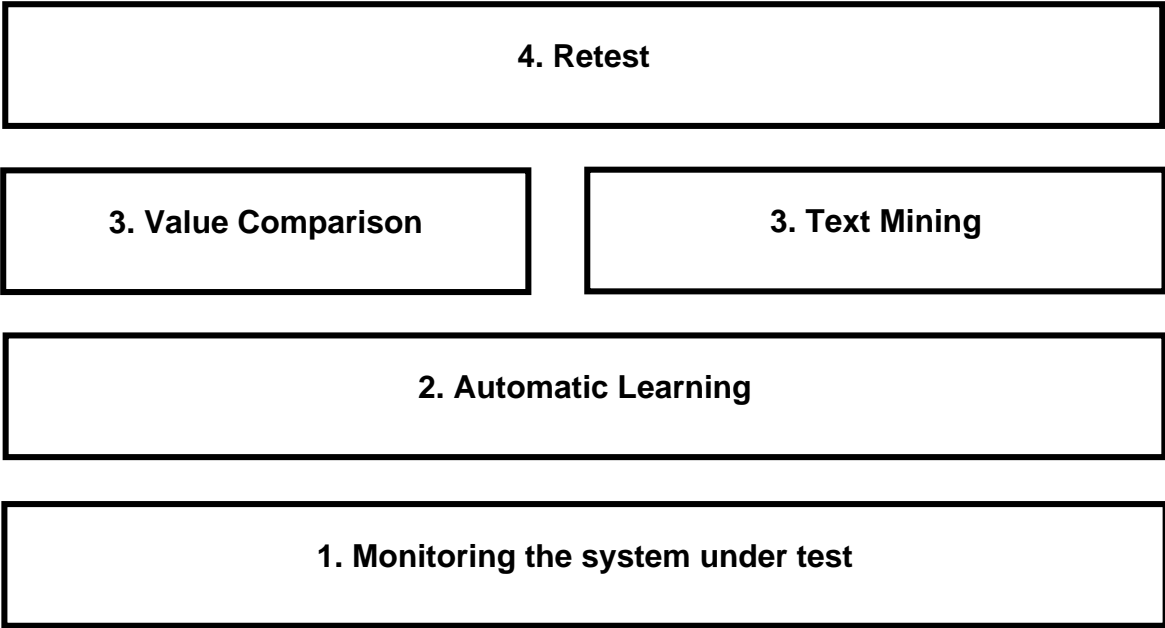


Figure 4.1: Functional architecture for a retesting based security error detection

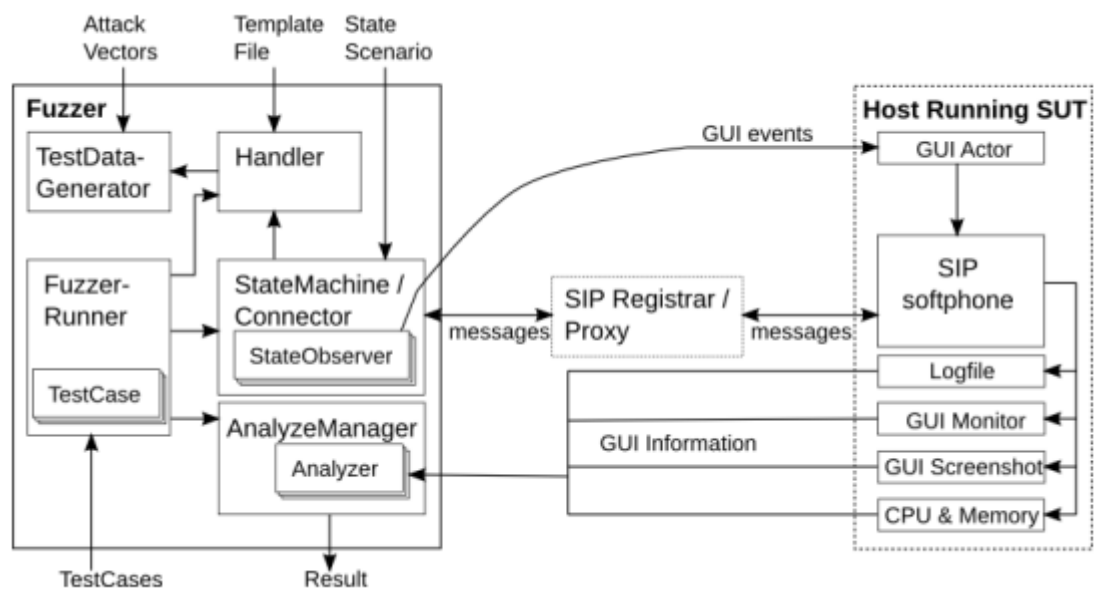


Figure 4.2: Infrastructure of the fuzzing framework [STP⁺11]

- TestDataGenerator
- Handler
- FuzzerRunner
 - TestCase / test Scenario
- StateMachine
 - StateObserver
- AnalyzeManager
 - Analyzer

The Analyzer are (together with the AnalyzeManager) the important part in this thesis because they are used to monitor the SUT, detect and reveal security vulnerabilities.

TestDataGenerator

Test data are specific input values that can be used within a test. For the generation of the test data two approaches are used: on the one hand there are randomized character- and integer values and on the other hand so called attack vectors are used. These attack vectors are input values that are known to cause or rather reveal possible attacks like SQL injection or XSS.

Examples for such attack values are: [Cla12]

- '
 - ' or '1'='1
 - ' or 1=1;–
 - <script>alert("1")</script>

Handler

The Handler is used for generating the messages that will be sent to the SUT. It takes a template of the messages to send as input. These template files can for example be provided in XML-format. This template also contains placeholders for the input values that should be fuzzed. During a test run, these placeholders are replaced by values that are created by the TestDataGenerator.

FuzzerRunner

The job of the so called FuzzerRunner is to control the test execution. Therefore it cooperates with the Handler, the StateMachine and the AnalyzeManager so that for each test case all the tests can be created by the Handler, executed by the StateMachine and analyzed by the AnalyzeManager. A test case (also called test scenario) is defined in template files and can be tested with the help of the fuzzer. A test is the test case in combination with specific, generated input values that are used instead of the placeholders in the template files.

TestCase / test scenario

A test scenario is also called a test case in context of this thesis and it is a sequence of different steps that are taken in order to test a specific functionality of the SUT. For example: The specific functionality to test is that a timing services responds with the correct time when it receives a request. The test scenario is to send a request to the timing service. A more complex example: The functionality of a user sending an email via an mail application should be tested. The test scenario is that the user logs in with username and password first and and if the login is correct, an email is sent.

In contrast a test is an execution of a test scenario by using specific test data. An example of a test for the mail application test scenario before is that the username "admin" and the password "secret" is used.

For the purpose of the fuzzer, the test scenario (or test case) is represented by one or more template files (that are used by the Handler described before) with placeholders for the input values. In one test run one test scenario is tested. For this test scenario, multiple tests are created by replacing the placeholders with different malicious input values created by the TestDataGenerator. For each of these tests the created requests are sent to the SUT and then its state is observed.

StateMachine

The StateMachine is used, when for testing purposes different states have to be managed. For example: If the sending of mails via an email application has to be tested, then before the sending of the mails the user has to be logged in. So at first the StateMachine would handle the login process, where valid user data is sent to the application. Afterwards the sending of the mails can be tested with the test data generated by the TestDataGenerator and the Handler as mentioned before.

In [TSH⁺10] it is described how the StateMachine was used for testing the SIP protocol.

AnalyzeManager

The purpose of the AnalyzeManager is to generate the final result. This is the rating of the current test. Therefore it collects the data of all the configured Analyzers and cal-

culates based on this data how likely it is that a security vulnerability has been detected.

4.1.2 Analyzer

The Analyzers can be seen as sensors that capture the current state of the SUT and the properties of the test itself. There are Analyzers that check if the application is still reachable after a test run with the help of TCP or UDP connections, Analyzers that collect the response of the application, Analyzers that monitor the changes in the logfiles and so on.

Each Analyzer rates for itself, based on the data it gets from the SUT, how likely it is that an error has occurred. For example: If the SUT is not reachable any more, then the Analyzer can be sure that it has crashed and an error has occurred. If in the logfile of the SUT the word "ERROR" occurs that does not necessarily mean that a security error has been found, but it could be a hint. In both cases the Analyzer has to rate its findings and report them to the AnalyzeManager.

4.2 Problem with detection of security errors

According to Thompson, security errors are unexpected side effects that are not defined or demanded by any requirements. [Tho03] [Sch13] This is also represented in figure 4.3: Not the specified behavior is implemented but another behavior that overlaps with the wanted behavior, but has also missing functionalities on the one hand and additional side effects on the other hand. [Tho03]

The problem is that the common software testing process is verifying if the wanted behavior, defined by the requirements, exists in the application and "not the absence of additional behavior". [Tho03] Therefore functional tests can not be used to reveal these hidden side effects, because there exist no requirements that can be tested. [Sch13] The automatic security testing tool could help to reveal such hidden side effects.

4.3 Monitoring the system under test

The Analyzers are used to monitor the SUT. Based on the system that is tested the Analyzers that are useful have to be chosen. [TDM08] For example: If an application has no graphical interface, then the GUIAnalyzer will not help. If a SQL injection should be detected, then the ResponseAnalyzer could be very important for detecting this error.

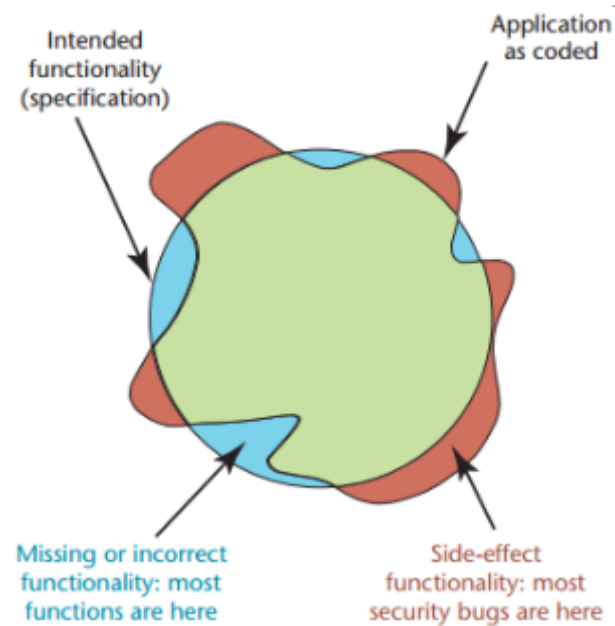


Figure 4.3: Deviation of specified behavior and actual implemented behavior including unwanted side effects [Tho03]

During the execution of the fuzzing-test the SUT can and should be monitored. In the following it is described what hints could be found in the monitored data that could help to reveal vulnerabilities.

Indications of security errors can for example be found in:

- Graphical User Interface of the SUT
- Systemload
- Logfile
- Network traffic
- TCP/UDP reachability
- Opened files
- Response message of the SUT
- Response time
- Logfile

- Spawned processes
- Database access
- Microsoft Windows Registry
- Output of a debugger

4.3.1 Graphical User Interface of the SUT

The usage of a GUI-monitor was already described in [TSH⁺10]. It was installed on the SUT itself to detect, if new error dialogs popped up during the testing of the system because this could also indicate a security error.

There are different indications for error information that can be observed in the user interface: [STP⁺11]

- Are there new opened windows after the test like for example error dialogs?
- Are the new opened windows belonging to the tested application after the test? This can be noticed on the basis of the title of the window.
- Are there fewer windows open after the test? This could indicate a crash.

The Analyzer monitoring the GUI is called the GUIAnalyzer.

4.3.2 Systemload

It can also be monitored, how much CPU and how much memory is used by the tested application during a test, sent by the fuzzer. This can be used to detect DOS attacks that try to consume so many resources of the system, so that it can not respond any more. [TDM08] Example tools for monitoring are the Process Explorer (Windows) and ps (Linux). [TDM08]

The Analyzer monitoring the systemload is called the SystemloadAnalyzer.

4.3.3 Logfile

For each test sent to the SUT, the changes in the logfile for the tested application are also monitored. In the logfile there can be hints to detect if a security error was raised.

When testing an application it should always run in debug mode if available. In this mode, as much output as possible is written to the logfiles. The more information are in the logfiles the more likely it is, to find hints to security errors. [TDM08]

Different applications log different information in different detail to the logfiles. For example: When exceptions occur in an application, one possibility is to only log if the

```
[Sun Dec 16 20:54:15 2007] [notice] child pid 174 exit signal
Segmentation fault (11)
```

Figure 4.4: Logfile entry of an apache server after a child process has crashed [TDM08]

exception has led to an error state, whereas another possibility is to log the exception although it has been caught and handled by the application. Because the fuzzing framework should, as mentioned before, work for all different kinds of applications this has to be considered. Not every log entry that says "error", does automatically mean that an actual error that could not be handled by the application itself, has occurred. [TDM08]

An example of an error that can be detected by the analysis of the logfile is described in the following:

An Apache server normally listens at port 80 for requests and manages multiple child processes. When a request arrives at port 80, the server forwards it to one of the child processes. If a malicious input is sent to the apache server that causes the childprocess to crash, the server itself is not affected. The next request is simply forwarded to another child process. If not enough child processes are available, a new one is spawned. Because of this behavior this kind of crash can not be detected by a simple TCP request, because the apache server itself is still available. This event can only be seen in the logfile (also see figure 4.4). [TDM08]

Another example that can be typically be found in server logfiles: When the server wants to consume more resources than there are available on system, this also leads to an error. [TDM08]

As a summary, the following things can be checked in a logfile:

- Are there any keywords, like "exception", "error", "segmentation fault", etc. in the logfile? (But it also has to be kept in mind that not each mentioning of an error in the logfile means that an actual error has occurred.)
- What are the differences of the log entries, when different input values are sent?

4.3.4 Network traffic

After monitoring the network traffic, besides other information, the message sent as an answer by the application can be extracted. [TDM08]

Things that can be checked to reveal security errors are:

- If the purpose of the current test is to reveal a XSS attack and the sent attack value is part of the response message, then this could indicate that a reflected XSS

vulnerability was found. [ZHYS11] [Bac10]

- The tests themselves and the response of the SUT can be clustered, so that all messages that provoke the same response are pooled. [CWKK09].

4.3.5 TCP/UDP reachability

After each test, it has to be checked, if the system has not crashed during the test. So the fuzzer tries to establish a TCP or UDP connection to the SUT. If this is not successful, then it can be assumed that the SUT is not alive any more and a security error has been found.

The Analyzer that monitors, if the system is reachable, is called the ReachabilityAnalyzer.

4.3.6 Opened files

Another aspect of the system that can be monitored, are the files that the application tries to open (no matter, if they could be opened or not). By inspecting the files and the filenames, it could be possible to detect directory traversal attacks. [TDM08]

Things that can be checked to reveal security errors are: [TDM08]

- When different input values are sent to the application, are there different files that are opened?
- Are there any input values that also appear as part of the filenames?

Example tools to monitor the opening of files are Filemon (from Microsoft) and strace (available on Linux). [TDM08]

4.3.7 Response message of the SUT

The response message is the answer the automatic testing tool gets from the SUT, after sending a request. If there is no answer, the possibility is high that the SUT has crashed and so we can be sure that a security error has been found. [TDM08]

The Analyzer monitoring the response message is called the ResponseAnalyzer.

4.3.8 Response time of the SUT

The response time is the time between the sending of the request message to the SUT and receiving the response message from the SUT. If there is no response (that is, when the response time is higher than a configured timeout), this could indicate a security error because the SUT has probably crashed.

The Analyzer that monitors the response time is the ResponseTimeAnalyzer.

```
[cmiller@Linux ~]$ strace -ewrite testprog
...
write(3, "\23\0\0\0\3select * from help", 23) = 23
```

Figure 4.5: SQL statement sent to a database, monitored by strace [TDM08]

4.3.9 Spawned processes

The number of new processes spawned by the application can also be monitored. [TDM08] The example of the apache server explained earlier can also be used here: When observing the processes, it could be detected that one of the child processes has crashed.

Tools to monitor, if new processes are created, are the Process Explorer (Microsoft) or strace (Linux). [TDM08]

4.3.10 Database access

It is possible to monitor the interaction of the tested application with a backend database (if one exists). Afterwards the SQL statements sent to the database and the response of the database can be inspected. A tool to do so, is strace as can be seen in figure 4.5.

4.3.11 Microsoft Windows Registry

In operating systems from Microsoft, also the registry entries can be checked. There for example it can be inspected if the privilege level of the application has changed during a test. One tool to detect changes in the registry is the Registry Monitor from Microsoft. [TDM08]

4.3.12 Output of a debugger

The application itself can also be supervised by a debugger. So the debugger is informed about exceptions or other events that occur during the test. [TDM08]

With a debugger it is much easier to find the point in the source code that has caused the error. [TDB12] A disadvantage of using a debugger on the other hand is that many more resources are needed for the test, so that the debugger "may slow down the SUT up to fifty times". [TDB12]

There are also more advanced methods than simply attaching the debugger to the application: Library inception, Binary simulation, source code transformation and virtualization. [TDM08] They are not further described in this thesis, because in some cases they need massive changes in the application itself and the idea of the fuzzing framework

that is supposed to be improved, is to do black box tests that are easily adaptable to all kinds of applications.

4.4 Attack detection by automatic learning of IDS

Another idea for improving the results of the fuzzer is to introduce a phase at the beginning of the automatic security test where the fuzzer learns, what valid behavior of the SUT looks like. With this knowledge, it is able to distinguish later between behavior of the SUT that is normal and behavior that could indicate a found security vulnerability. This is also described in a paper about the detection of vulnerabilities in VoIP softphones. [STP⁺11]

Concepts for automatic learning and for error detection are to be found in the field of intrusion detection systems (IDS). [AG10] [CGG⁺09] In a learning phase, valid packets are received, analyzed and stored first. They are the model for all future packets so all divergent behavior is detected as an anomaly. A similar mechanism can be used for security tests – the behavior of the system under test in normal cases and in error cases is compared.

Yu et al. describe in a paper that IDS are systems that try to detect "attempts to compromise the confidentiality, integrity, availability of a resource, or to bypass the security mechanism of a computer or network system". [YT11] The goal of an IDS is therefore, to monitor the behavior of a system and to find all attempts of an intruder that could jeopardize the security of the system. [YT11] [SP10]

4.4.1 Methods for intrusion detection

Basically there are multiple methods of how an IDS tries to detect an intrusion which can be used for security tests: [SP10] [GLT10]

- **Misuse detection** - It is known, what an attack looks like. [SP10]
 - **Predefined patterns or signatures** - In a so called knowledge base, the typical sequence of actions for different attacks is stored. This sequence is called a pattern or a signature. If the IDS recognizes the same sequence of actions while monitoring the system, it assumes that a security attack has been found. Attacks, where no signatures are stored, can not be identified. [YT11] [YMTT05]
 - **Build patterns or signatures** - Some IDS use data mining techniques to build new patterns and to put them into the knowledge base. Patterns that are not in the knowledge base can still not be identified. [YT11]
- **Anomaly detection** - Yu et al. state that "Anomaly detection is concerned with identifying events that appear to be anomalous with respect to normal system

behavior”. [YT11] So called profiles are generated that represent the normal behavior of the user, the network and so on without an intruder present. If activities happen on the system that are not assignable to a profile, then the IDS suspects an intrusion and reports it. [YT11] [SP10] [GLT10] Details to anomaly detection can be found in subsection 4.4.2.

- **Specification-based detection** - The system has knowledge of the normal behavior of the system. This knowledge is defined by security experts. All behavior that is not similar to the defined, normal behavior is identified as an attack and reported. [GLT10]

Nowadays in industry most IDS use misuse detection for identifying attacks. [SP10] [TSG10] The reason for this development is that misuse detection models have a ”predictability and high accuracy”. [TSG10] In the field of research, anomaly detection is favored, because of its ability to reveal unknown attacks. [TSG10]

Ideally all three of these methods are used in combination to improve the ability of the automatic security testing tool to reveal vulnerabilities because each of them has other advantages:

- The concept of *misuse detection* has the advantage that it is feasible for all kind of applications. It will be implemented in the framework by using the Analyzers and the indications for security errors.
- The concept of *specification-based detection* has the advantage that the system knowledge is customized by security experts for the tested application and therefore it is likely to find more vulnerabilities than *misuse detection*. The disadvantage on the other hand is that the customizing process is additional work that has to be done, before the actual testing can start. Also the need for expert knowledge stands in contrast to the easy usage of the tool for the developers. This concept was already used successfully in the fuzzing framework for testing some applications, but because the knowledge is not reusable, this method is not an appropriate approach for this generic fuzzing tool. Therefore this method is not further described in this thesis.
- The concept of *anomaly detection* has the advantage that it helps to automatically gain knowledge about the normal and abnormal behavior for any kind of system. This method is described in detail in the following sections.

4.4.2 Anomaly detection

Anomaly detection is based on the assumption that any attack provokes an abnormal behavior on the SUT. [GLT10] The goal of anomaly detection is therefore to detect

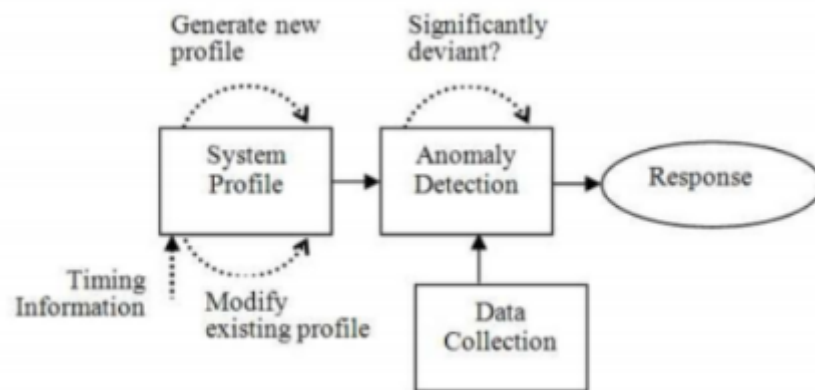


Figure 4.6: Structure of an anomaly detection system [GLT10]

abnormal activities on the system with the knowledge of normal reactions. With this approach also new kind of attacks can be found, because they diverge from a normal behavior. [YT11] [GLT10]

This can be compared to black- and whitelisting as it is known for input validation. Blacklisting knows invalid inputs that have to be filtered, whereas values that are not on the blacklist are automatically allowed. The problem is that a possibly infinite list of unwanted values exist, which can not all be predefined. Whitelisting on the other hand, has knowledge which pattern the input has to have and all other values are declined. If possible, whitelisting is preferable to blacklisting. [Hus04]

Structure of anomaly detection systems

The typical structure of an anomaly detection system can be seen in figure 4.6.

It consists of four components :

- **Data Collection** - The data collection module monitors and stores all the information of normal activities of the system. [GLT10]
- **System Profile** - The system profile module creates so called system profiles that represent the normal behavior, based on the collected data. [GLT10] The data of a system profile of a user could for example include: the applications that are normally used, the number of open windows, the websites that the user normally visits. [PA10]
- **Anomaly Detection** - The task of the anomaly detection module is to decide, if the monitored behavior is similar to the normal behavior or should be classified as a found anomaly. It gets the information what normal activities are from the data collection module and the system profile module. [GLT10]

- **Response** - If the anomaly detection decides that an anomaly has been found, a response is sent to the system operator, containing information about the possible attack. [GLT10]

Problems of anomaly detection systems

The problem with anomaly detection is to distinguish, if a monitored behavior is similar to the behavior that is known as normal or if it is already abnormal. [YT11] Also if the training data do not cover the full range of normal behavior, then false alarms will be raised in case of not learned, normal activities. [YMTT05] Therefore the false-positive rate will be higher than the false-positive rate of pattern matching. This is also a known problem that existing IDS have: either they report too many false alarms (and the system operator will ignore them in the future which leads to the problem that real attacks are missed) or they miss real attacks (and this could lead to the problem that the IDS loses the system operators trust). [YT11] [YH12]

Another problem are false-negatives of reported attacks and there are two causes: One problem is that wrong information is learned and therefore accepted as normal behavior. All activities of intrusions that are so mistakenly learned by the IDS will never be recognized. The second problem is that not all necessary components are monitored. For example: if only the CPU consumption is monitored, but not the memory usage, then attacks that have only an effect on the memory but not on the CPU, will never be detected. [DV05]

Methods to improve the quality and quantity of intrusion reports

The following methods exist to improve the quality (which means to reduce the number of false-positives) of the IDS alarms:

- **Improvement of learn/check method** - The methods that identify the normal behavior and check if the monitored behavior is similar to an already known one, have to improve. [YT11]
- **Additional program** - An additional program has to be introduced that takes the output of the IDS and tries to filter the false-positives (if possible) or adds additional information to the IDS report so that it supports the system operator. [YT11]

These methods exist to improve the quantity of the IDS alarms:

- **Alert correlation** - Khakpour et al. state that "Alert correlation is a technique to extract attack scenarios by investigating the correlation of intrusion detection systems alerts". [KJ09] Alarms that belong to the same attack are grouped, so the system operator has less alarm messages to check. [KJ09]

- **Alarm filter** - Data mining techniques are used to help the administrator to distinguish between false alarms and actual abnormalities. So the number of false positives can be reduced. [MCF10] [YT11]
- **Event classification** - Intrusion detection systems have to classify the inspected actions as harmless or as intrusion. To reduce the number of false alarms, the relevant information has to be monitored and the calculation of the overall result has to improve. [KMRV03] [YT11]

4.4.3 Machine learning concepts

Machine learning is a general concept that is used in different operation fields (like for example robotics, medical diagnosis or artificial intelligence), where the generation of new knowledge is needed. [VK12] Anomaly detection systems need to learn how the normal activities of a system look like.

Basically there are two methods of machine learning: [JZ06] [YMTT05]

- Supervised learning
- Unsupervised learning

In the following subsections these two methods are explained with an example from the field of operation of IDS.

Supervised learning

Supervised learning ensures that the learning phase of the IDS to learn the normal activities of the system is controlled by a tester. This person checks that no anomalous data is hidden in the training data before the learning process starts. The problem is that the system can change over time and therefore the learned information does not necessarily be valid any more. [JZ06]

Suppose an IDS that should check a network for anomalies. Eventually the network will change, because new components or services are added. If not another learning phase is done after each change in the network, this will lead to a high number of false positives. Most of the network intrusion detection systems nowadays use supervised learning. [JZ06]

Unsupervised learning

The idea of unsupervised learning is that the IDS can adapt to new circumstances in the system itself without any help. In this case, the training data do not have to be free of

anomalies, because with the help of Data Mining outliers are eliminated. [JZ06] Another possibility to get automated generated training data is to use the generated output of a misuse detection system. [YMTT05]

4.4.4 Concepts of abnormal behavior checking

IDS also need a mechanism to decide, if the monitored behavior is similar to the learned, harmless behavior.

For IDS there exist multiple concepts of how abnormal behavior can be detected: [GLT10]

- Advanced statistical models
- Rule-based models
- Learning models

As Thompson stated in [Tho03], security vulnerabilities are unwanted side effects in an application. The concepts for detecting this unwanted and abnormal behavior can be used in security testing and are described in the following sections.

Advanced statistical models

A statistical analysis is used to distinguish between a normal and an abnormal behavior. The statistical model is used to decide, if the monitored behavior leads to an adaption of the learned profiles (which is a kind of unsupervised learning) or to a reporting to the system operator. [GLT10]

In the learning phase various activities on the system are monitored like the CPU usage, the number of processes running and so on. So a range of normal values is defined. If after the learning phase the inspected values are outside the range, then the IDS suspects that an intrusion has been detected and reports it. [GLT10]

Rule-based models

Based on the learned data, rules are derived. These rules are later used to decide if the monitored activities are normal activities or not. Based on the violated rules also a score can be computed. If this score is higher than a defined threshold, the activities are reported. An example for rules would be, which computers typically communicate with each other. Another example is that after monitoring the sequence of activities a user has performed, it is calculated, what the next step of the user would typically have to be. [GLT10]

The problem of this approach is the higher number of false-positives that are created if the system changes, because for example new applications or computers are introduced to the network. [GLT10]

Learning models

For this approach a learned model that is generated with the help of training data is used. [VK12] The two kinds of learning models, namely supervised and unsupervised learning, have already been described.

4.4.5 Summary

In this section it has been described, which concepts exist for IDS to detect attacks. IDS learn the normal behavior of systems and by detecting deviations of the monitored behavior to the normal behavior they can find and report possible attacks.

The idea is that these concepts also help the automatic security testing tool to reveal security vulnerabilities because security errors are also abnormal side effects in an application. [Tho03] Therefore the normal behavior of the tested application can be learned and any abnormality could indicate a found vulnerability.

There are three types of methods that help to find intrusions: Misuse detection, anomaly detection and specification-based detection. The concept of misuse detection is used for the fuzzer in the form of the Analyzers. Also anomaly detection will be used because anomaly detection has the ability to detect new kinds of attacks for any tested application. Also the concept of supervised learning is applicable, because the behavior of an application does not evolve during a test run.

Also the structure of an IDS along with the typical problems and solution concepts were described.

4.5 Automatic learning for security tests

In the following it is described how the concepts existing for IDS can be adapted, so that they help to improve the fuzzing framework.

Every test run of the fuzzing framework will be composed of two phases:

- Learning phase
- Fuzzing phase

In order for the fuzzer to learn the normal behavior of the SUT in the learning phase, it is also needed that a normal behavior is created on the SUT so that can be monitored. This can be achieved with the help of valid test cases that are introduced in the following section.

4.5.1 Valid cases

In valid tests the fuzzer sends valid input values to the SUT and the SUT responds in a specified and expected way. A valid test does not have the goal reveal a security error. For example: When testing a valid test, the SUT will not crash and so the Analyzer that checks if the tested service is still available by trying to create a TCP- or UDP-connection, will always get a response. This is also called the connectivity check. [TDM08] [SGA07] According to [TDM08], valid tests can also be used for examining the SUT. If a valid test case is sent after each fuzzed test case it can for example be checked if the system is still available and reacts as expected. [SGA07] [TDM08]

Possibilities to create valid inputs are: [TDB12]

- **Collect typical input data** - This approach is needed, if complex data has to be used, like for example image files, text files, real-life data of a database and so on.
- **Formal Model** - A model that defines how the input has to look like is created. Based on this definition, input data is created.
- **Monitoring** - The fuzzer can be used as a proxy between the SUT and the opponent of the SUT. For example, if a webserver should be tested, then the webserver is the SUT and a client that sends requests to the webserver is the opponent. The fuzzer monitors the communication and learns possible valid input values.

4.5.2 Learning phase

Additionally to the already existing fuzzing phase in the fuzzing framework, a so called learning phase is introduced at the beginning of a test run: In this phase, the fuzzer runs multiple different valid cases and the behavior of the SUT is monitored. So the fuzzer learns, what a normal and harmless reaction of the SUT looks like. This is done with the help of the Analyzers that monitor the system.

The learning phase is important for the fuzzer to distinguish between normal behavior and vulnerabilities. The valid cases of the learning phase have an influence on the results of the fuzzer. Normal behavior that is not learned during the training phase, can not be recognized as normal during the fuzzing phase. [CWKK09] So it is possible that the fuzzer reports false positives during the fuzzing phase, because a normal reaction of the SUT that the fuzzer does not know, could be classified as malicious.

4.5.3 Fuzzing phase

After the end of the learning phase, the so called fuzzing phase starts. In this phase the potentially malicious input is sent to the SUT and the behavior is monitored. [BBGM12] At each test, the state of the SUT is monitored by the Analyzers and then different decisions have to be made to decide whether the found behavior indicates a possible

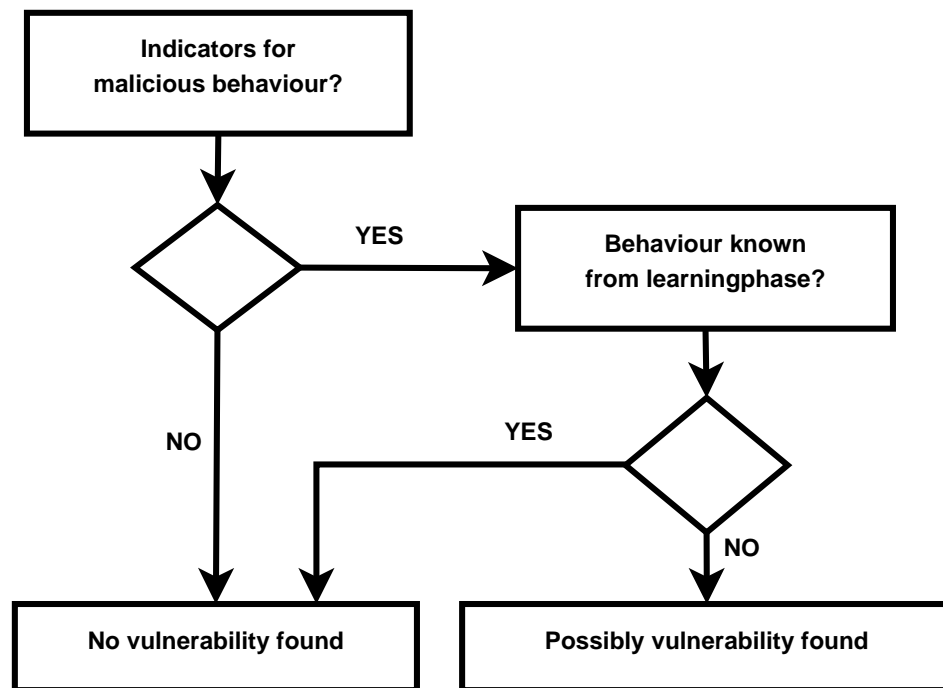


Figure 4.7: Decisions made during the fuzzing phase

vulnerability or not. This can also be seen in figure 4.7.

4.6 Data Mining and Text Mining

According to Witten "Data Mining is the process of discovering patterns in data". [WFH11]

These found patterns can be used for instance to extract new information or to make predictions. [WFH11] Furthermore outliers of data can be found or big amounts of data can be aggregated so that they are easier to analyze. [HKP12]
The goal of Data Mining is to turn "a large collection of data into knowledge". [HKP12]

4.6.1 Functionalities of Data Mining

There are different kinds of functionalities in Data Mining to extract patterns: [HKP12] [Gup11]

- Classifications
- Clustering analysis

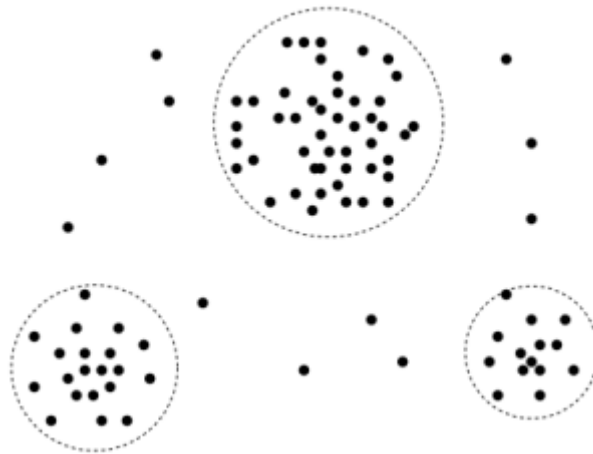


Figure 4.8: Graphical representation of clustering [HKP12]

- Outlier analysis
- Associations

Classifications

Han et al. state that "Classification is the process of finding a model [...] that describes and distinguishes data classes or concepts". [HKP12]

At the beginning of the classification process models are created with the help of training data. The models can be seen as categories that describe the data that is put into these categories. After the trainings run, the actual data are classified and put into the different categories. [HKP12]

An example for a classification is: *tests that have found a vulnerability* and *tests that have not found a vulnerability*. First training data that contain data sets that have found vulnerabilities and that have found no vulnerability, are needed. Then the data is analyzed, to find the properties that are typical for either of these groups. An example of a property for the category *tests that have found a vulnerability* is that the word "segmentation fault" was found by the LogfileAnalyzer. The actual data is then also examined. All data, where the word "segmentation fault" was found by the LogfileAnalyzer, are then also classified as *tests that have found a vulnerability*.

Clustering analysis

Clustering is similar to classification, with the exception that no training data is given. The clustering process only knows the actual data that has to be categorized and put into so called clusters. So the analysis process has to examine the given data to decide, how similar the single data sets are to each other. These found datasets that are similar

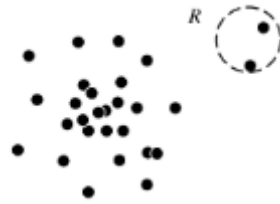


Figure 4.9: Graphical representation of outliers [HKP12]

to each other, are summarized in so called clusters. [HKP12] A graphical representation of clustering can also be seen in figure 4.8.

Outlier analysis

Normally one step in the Data Mining process (the data cleaning step) is to remove data from the data base that does not fit the general behavior of the other data sets. This is the so called "noise" and is seen as a faulty dataset. However in some questions these outliers are especially interesting. An example would be a credit card purchase, where a purchase done is more expensive, than the usual ones. [HKP12] A graphical representation of clustering can also be seen in figure 4.9.

Associations

The goal of association rule mining is to find patterns that frequently occur together. An example would be that a supermarket analyzes, what kind of products are typically purchased together. [HKP12] [Gup11]

4.6.2 Functionalities of Data Mining for the fuzzer

To improve the fuzzing framework, the ability of Data Mining to aggregate data shall be used. After the test run, the fuzzer reports all the tests with the results calculated by the AnalyzeManager to the tester. The result is a value between 0.0 (the fuzzer could find no indications for a security error) and 1.0 (the fuzzer is sure that a vulnerability has been found).

The tester has to check the test cases to review the results of the fuzzer. Possibly there could be lots of test cases where the fuzzer is neither absolutely sure that a vulnerability (1.0) was found, nor that it is a harmless test case (0.0). To simplify the work of the tester, similar test results shall be grouped. So not every test of this group has to be checked individually, but only one is representative for the others.

In the following all the functionalities of Data Mining are rated, based on how useful they could be for the fuzzer:

- **Clustering analysis** - Clustering analysis is the one that fits best for the purpose of the fuzzer: All the data from the Analyzers is given and then the similar cases are aggregated, so the tester does not have to review every one of the tests.
- **Classifications** - Classification is also useful, but the only training data that are available, are that of the trainings phase of the fuzzer, which only contains valid data. The data of fuzzing phase can only be classified as *valid data* or *not valid data*.
- **Outlier analysis** - An outlier analysis is not reasonable for the fuzzer, because all of the test cases could possibly reveal a vulnerability - not only the outliers. If the outliers can not be assigned to a cluster, then they will also be presented to the tester. But there will be no data cleaning done by the fuzzer, unlike it is usually done in the Data Mining process. [HKP12]
- **Associations** - Association rules are not useful, because at each test, the Analyzer only has knowledge of the information of the data of the learning phase and the information of the current test case. What could be useful is that after the whole test run of the fuzzer, all the generated data are checked again, but this is outside the scope of this thesis.

4.6.3 Concepts of Text Mining

A special case of Data Mining is Text Mining that tries to extract useful information from text by finding patterns. [HKP12] [FS06] Unlike Data Mining, Text Mining does not use a database as data source but unstructured documents containing text.

In data mining systems so called background knowledge is used to improve the results of the data mining process. This background knowledge contains information which words or properties are important in a specific domain. [FS06]

Concepts of Text Mining that could be interesting for the fuzzer are explained in the following.

Usually the first step in Text Mining is the preprocessing phase, where the unstructured data found in a document is converted into a representation, where further processing is possible. [FS06]

Preprocessing operations are for example:

- **Bag-of-words** - The text is split up into individual words (so called *features*) and so the so called *bag-of-words* is created. This is needed to have a representation of the text where further processing can be done. The existing individual words are rated. There are different possibilities how the rating can be done. One example would be a rating based on the number of occurrences of each word in the bag-of-words. [FS06]

- **Stop words removal** - Stop words are words like *of, the, and, with*. They can be removed, because they do not represent the actual meaning of the text. [BK10] [GS09] [FS06]
- **Stem words** - Stem words are words that represent the root form of a word. For example, the stem word of *playing* and *played* is *play*. Words in the text can be reduced to their stem word, because the meaning of the words is retained, but the number of words existing is reduced. [GS09]
- **Case folding** - All words in text are either transformed into lower or upper case words. So words that are written in different forms are treated the same. [AHC10]

After the preprocessing processes, the phase starts that extracts the patters. The main techniques for Text Mining are: [BK10] [FS06]

- Keyword extraction / text indexing
- Document sorting
- Text streams

These are explained in the following subsections.

Keyword extraction / Text indexing

A keyword is a single word or a group of words that is part of a text. The goal of keyword extraction is to find words (keywords) that represent the main aspects of the content of a document. One approach is to determine the frequency of words. [BK10]

Steps of keyword extraction: [BK10]

- **Candidate keywords** - At first the bag-of-words is created and then the stop words are removed as described before.
- **Keyword scores** - The extracted keywords are rated, based on how often they occur and how long the word is.
- **Extracted keywords** - The best rated keywords are selected to be the keywords of the examined document. For example, the best third of the keywords is chosen.

One area of application of keyword extraction is the classification of spam mail so it can be filtered. The received mails are classified in the two categories "spam" and "not spam". To do so, a supervised learning method is used. At first training data is used that contain spam mails and legitimate mails that are already classified the right way. [BK10]

Document sorting

The goal of document sorting is to receive a lot of documents as input and then to categorize them. This task is like the classification and clustering task. The documents can also be assigned to multiple categories. Furthermore a so called *soft ranking* can be done. This means that the document is assigned a value between 0.0 and 1.0 to describe how well it fits into a certain category. This value is the categorization status value (CSV). If a soft category ranking has to be converted to a hard category ranking (which is a binary ranking, where a document is either part of a category or not), then a threshold has to be defined. [FS06]

Text streams

Text streams are a list of texts (in the form of documents or messages) that are created over a period of time. The goal when observing text streams is for example to find out, if the topic of the texts changes as time passes. [BK10]

4.6.4 Concepts of Text Mining for the fuzzer

In the fuzzing phase the Analyzers have to decide for the current test, how similar the behavior of the SUT to the behavior in the learning phase is.

Two of the Analyzers get text as input:

- **ResponseAnalyzer** - The ResponseAnalyzer monitors the response message of the SUT.
- **LogfileAnalyzer** - For every test the changes in the logfile of the application are monitored by the LogfileAnalyzer.

For these texts, captured at every test, the decision has to be made, how similar they are to the texts of the learning phase and if indications for an error can be found. This can be achieved with concepts of Text Mining.

The following concepts could be useful for the fuzzer:

- Background knowledge
- Keyword scores
- Stem words
- Case folding
- Document sorting

Background knowledge

The background knowledge contains domain specific knowledge. The goal of the fuzzer is to find indications for security errors in the texts monitored by the ResponseAnalyzer and the LogfileAnalyzer as described before. Typical indications could be words like *segmentation fault* or *error*. These words need to be defined as background knowledge for the fuzzer.

Keyword scores

The concept of keyword extraction is not useful for the fuzzer because there are no test cases in the learning phase where it is known for sure that these reveal an error. In the learning phase of the fuzzer, only valid cases exist. So there is no data available, where keywords that indicate error, can be found.

The concept of rating the keywords is suitable for the fuzzer.

For example: If the word "segmentation fault" is found in a text, then it is very likely that a security vulnerability has been found. On the other hand, if the word "error" exists, it is not that likely, because the logfile could also contain the sentence "There was no error found."

To express the likelihood of an error a rating between 0.0 and 1.0 could be used for each keyword.

Stem words

The words defined in the background knowledge should be defined in the form of stem words. So it can be checked in the found texts, if words exist that also have these stem words and so new keywords can be found. For example: If the word defined in the background knowledge is *err*, then words like *error* and *messageerror* could be found and so new keywords can be defined.

In difference to the stem word concept the goal of the fuzzer is not to reduce the number of different kinds of words, but to identify many kinds of words as new keywords. So on basis of the defined keywords in background knowledge, new keywords and so new indications for security errors can be found.

Case folding

Case folding is also useful for the fuzzer, because when looking for indications for security errors it makes no difference if the words are written in lower case or upper case. For example: *Segmentation Fault* vs. *segmentation fault*.

Document sorting

For each received text (monitored by the ResponseAnalyzer or the LogfileAnalyzer) also a categorization has to be done. Either it is categorized as normal behavior (as learned

in the learning phase) or indications for vulnerabilities are found. Soft ranking is also useful in combination with the rating of the keywords as described before. Each of the keywords have a ranking between a value of 0.0 and 1.0. The Analyzer can calculate an overall ranking (between 0.0 and 1.0) based on the found keywords and report this value to the AnalyzeManager.

The AnalyzeManager receives a value between 0.0. and 1.0 from each of the Analyzers. For the calculated overall result also soft ranking is applicable. So the tester can sort the results based on their likeliness of a found vulnerability when checking them for true- and false-positives.

Chapter 5

Process driven approach for improving error detection precision

The goal of this thesis is to find a procedure that can reveal security vulnerabilities. In this chapter a retesting concept is elaborated that could help to improve the results (finding vulnerabilities and reducing the number of false-positives) of the fuzzing framework.

Retesting means that a test that has already been carried out, is done again (after some time). The tested system does not change. [Jac11] [Dan10]

In psychology test-retest methodologies are commonly used (for measuring the reliability of tests). [Jac11] [Dan10] In computer science retesting is also termed as regression testing. Retesting is applied for the testing of wafers which are always tested twice. If there are any differences in the two test runs, then the test is repeated again to find the errors. [KPP11b]

The procedure of regression testing is as follows: When a new version of a software exists, it is tested again, by using the test cases that were already used for testing the old version of the software. So it can be assured that no new errors have been implemented in the software and all functionality of the old version still works in the new one. [MAM11] [HO08] Regression testing is not truly retesting, because the test is not carried out on the same version of software.

The idea for this thesis is that if an abnormal behavior is inspected on the SUT, then the same test is carried out again (it is retested). If the found abnormal behavior occurs again, then this behavior is reported as a possible found error.

This procedure is needed, because the result of a test can be unexpectedly influenced by the testing environment. An example is a peak in the CPU- or network-load caused by a process other than the tested application. These changed circumstances would cause the fuzzer to rate these unexpected behavior as an error. By doing a retest and examining

these result, the decision can be made if a security error was found or if actually the testing environment was the reason for the inspected unusual behavior.

An approach of how this can be achieved, is described in the following sections.

5.1 Introduction into retesting of test cases for security tests

In the fuzzing phase, also called testing phase, of the fuzzer one test after the other is sent to the SUT. After each test the Analyzers monitor the state of the SUT. The collected data could for example be the log data or the response message. This was already described in the section before.

After each test it is checked, if any "abnormal" behavior can be found in this collected data. This means any behavior that differs from the behavior of a valid test case that does not exploit a security vulnerability. An example for an abnormal behavior would be that the time until an answer message is received is longer than it is typically for a valid test case or that no answer message is received at all.

To decide, what a "normal" behavior is (and to distinguish it from an "abnormal" one) it has to be learned, how the typical behavior of the SUT looks like. For example: If exceptions are used by the programmer of the application to handle certain events, then the debugger registers this behavior, but this does not indicate a security error. If a "divided by zero" exception is raised, this could either be an intended behavior of the programmer and harmless, because the exception is handled, or it could be an error, because it was not expected by the programmer and only provoked by malicious input. [TDM08]

Retesting should also help the automatic security testing tool to distinguish between a normal and an abnormal behavior: It provides indications, if the monitored behavior was caused by the executed test or if it was also influenced by other events in the testing environment. For example: If an abnormal response time is registered for one test and this test is done again then there are two possibilities: Either the response time is normal, than it is likely that the reason for the delay was not the test itself but for example a high network load, or the response time is again high, then this could indicate that the reason for the delay is actually the test. In the first case, retesting helps to prevent a false-positive, in the second case, an indication for a vulnerability was found.

In order to implement this actual retesting concept, the other concepts already described in chapter 4 have to be combined as can be seen in figure 5.1.

At first a *learning phase* is introduced in the beginning that should help the fuzzer to distinguish between tests that cause a security error and those that do not. In this

phase only *valid tests* are sent to the SUT that create normal behavior on the SUT. For each of these tests a *monitoring* of the behavior is done and this created normal behavior is learned by the Analyzers. [SHFG13]

In the second phase, the *testing phase*, malicious input is sent to the SUT. Again, the Analyzers *monitor* the behavior of the SUT. This collected data have to be examined to find evidence of security errors. There are different concepts of how indications for malicious behavior can be found: If the captured values are numeric (like the response time or the CPU usage), then a *comparison of the values* found during the test to the values found in the learning phase can be done. [Sch13] Different concepts and statistical methods exist of how this comparison can be done as described in [Sch13]. If the behavior is represented by text (like the response message or the data found in the logfile), then in some cases *Text Mining* will be needed to decide, how similar the found text is to the texts found in the learning phase. Also the counting of the words existing in the text or other simple methods could help to reveal vulnerabilities. For example: If the exploitation of a SQL injection vulnerability provides all credit numbers existing in the database than this response text is presumably longer than if only one credit card number is returned.

If the so found behavior differs from the normal behavior, then *retesting* can be used to decide if an actual security error has been found. After each test that has caused an abnormal behavior (like a longer response time than all the response times in the learning phase) a valid test and then the test itself is repeated again (retested). The information gathered in these three testing steps can be used to identify security errors. This concept will be explained in detail later in this section.

All these concepts, including the elaborated retesting concept, will be applied to the fuzzer later on.

5.2 Retesting concept

Like already explained in chapter 4 the goal of a fuzzer is to reveal security vulnerabilities by sending malicious input to the SUT. The concept of retesting should be discussed to help accomplish this goal.

5.2.1 General description of the retesting concept

The flow of the retesting concept can be seen in figure 5.2. A learning phase is needed in the beginning, where the fuzzer learns the normal behavior of the SUT, so it can distinguish it from the abnormal behavior. So the fuzzer sends multiple different tests that do not contain any malicious input, to the SUT and observes the behavior of the

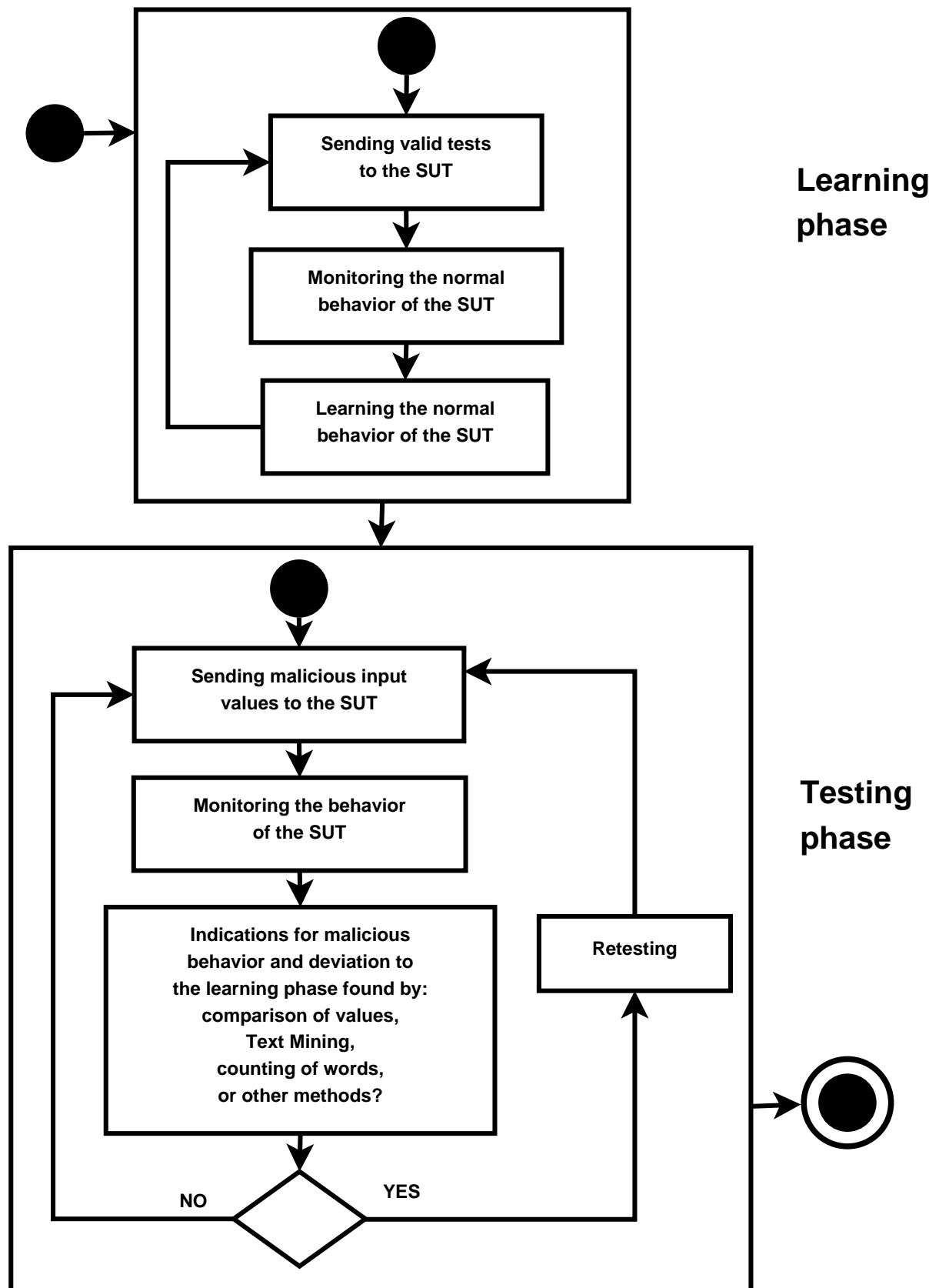


Figure 5.1: Steps for a retesting based security error detection

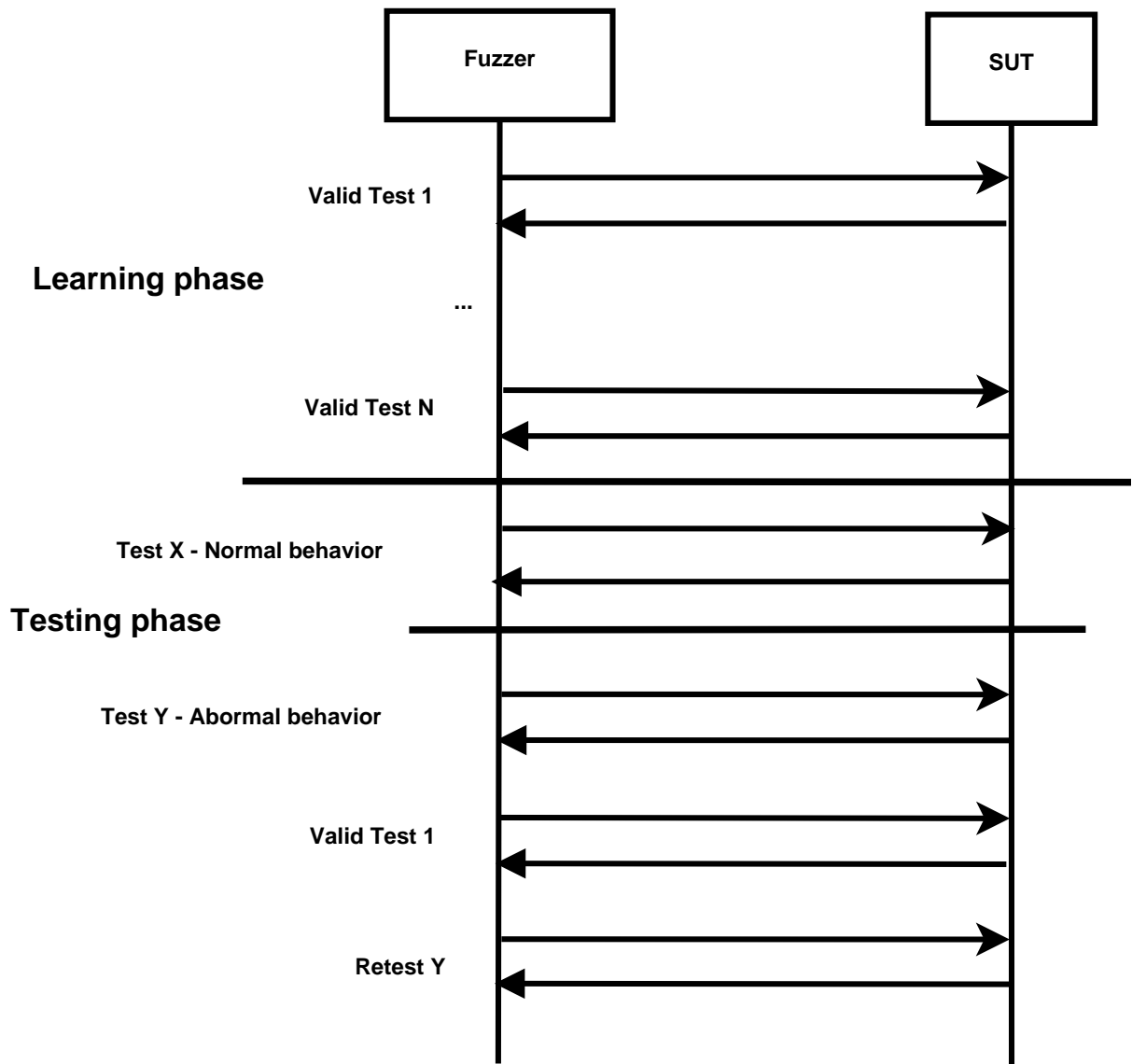


Figure 5.2: General concept of retesting

SUT with the help of the Analyzers. One example would be that the fuzzer has learned at the end of the learning phase that the response time of a valid test is always less than one second.

In the fuzzing phase the malicious input is sent to the SUT. In some cases (like test X in figure 5.2), the behavior observed on the SUT does not differ from the behavior of the learning phase. The test did not provoke unusual behavior on the SUT and so did not reveal a security error.

In case of test Y in figure 5.2 abnormal behavior (like a longer response time than was ever observed in the learning phase) did occur. So the retesting is done: After test Y, again a valid test is sent to the SUT and afterwards test Y is sent again.

When sending the valid test and test Y again to the SUT, four different cases are possible of how the behavior of the SUT could look like:

- Test Y (abnormal behavior) - valid test (normal behavior) - retest test Y (normal behavior)
- Test Y (abnormal behavior) - valid test (normal behavior) - retest test Y (abnormal behavior)
- Test Y (abnormal behavior) - valid test (abnormal behavior) - retest test Y (normal behavior)
- Test Y (abnormal behavior) - valid test (abnormal behavior) - retest test Y (abnormal behavior)

These four cases are discussed in the following section based on the metrics that are observed on the SUT.

5.2.2 Retesting the different metrics of the fuzzer

The following metrics are monitored by the fuzzer:

- Numeric
 - Response time - ResponseTimeAnalyzer
 - Systemload - SystemloadAnalyzer
 - Graphical User Interface of the SUT - GUIAnalyzer
- TCP/UDP Reachability
- Text-based
 - Response message of the SUT - ResponseAnalyzer

– Logfile - LogfileAnalyzer

In the following it is described, how the retesting concept could be used for the different metrics.

Response time

The response time is the time span between the moment, when a request is sent to the SUT and the moment when the response arrives at the fuzzer.

Retesting is necessary, when the response time of a test in the fuzzing phase is longer than the response time of the tests in the learning phase. Suppose that normal response time (during the learning phase) was less than one second. If the response time in the fuzzing phase is for example 5 seconds, then this is registered as abnormal behavior and so the retesting process starts: A valid test is sent to the SUT and afterwards the test again. For each of these two tests, the response time is monitored.

Four different cases can occur:

- **Valid test and retest: normal** - In this case, both the valid test and the retest respond in normal time (less than one second). This can be seen in figure 5.3. Such a behavior indicates that at the time where the test was done the first time, a network overload or any other reason for the delay occurred that was not caused by the test. If the test had been the reason for the slowdown, then this behavior would have been observable again at the retest - what was not the case. Therefore the result is: **No Error**.
- **Valid test: normal, retest: abnormal** - In this case, the response of the valid test is as fast, as it was in the learning phase, whereas the response time of the retest is again abnormal. This case can be seen in figure 5.4. In contrast to the case before, it is very unlikely that the reason for the delay is a network overload because the response time of the valid test was normal. So the result of this case is: **Error**.
- **Valid test: abnormal, retest: normal** - The response of the valid test takes longer, as it had taken in the learning phase, whereas response time of the retest is faster than when the test was done the first time, so that it is as fast as the tests in the learning phase. This is also pictured in figure 5.5. The reason for the delay of the first test can not be the test itself, because then the retest would also be slow. A reason for the occurrence of this case could be that there was also a network overload during the test and valid test that was cleared when the retest was done. The result of this case is: **No Error**.
- **Valid test and retest: abnormal** - After the long response time of the test, also the response time of the SUT during the valid test and the retest is longer

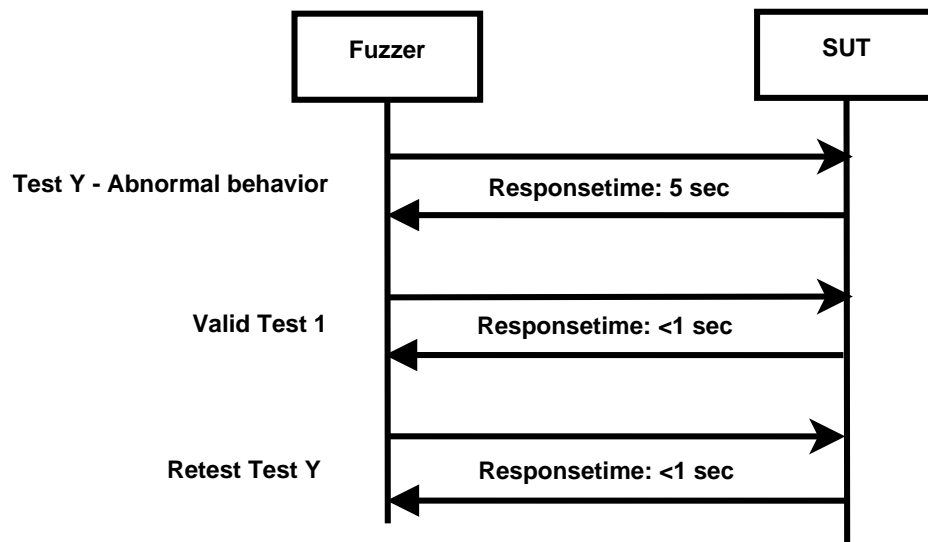


Figure 5.3: Test Y (abnormal behavior) - valid test (normal behavior) - retest test Y (normal behavior)

than the response times in the learning phase. This can also be seen in figure 5.6. There could be two reasons for this behavior: One possibility is that the first test is the reason for this delay. The test revealed an error in the SUT that caused the SUT to slow down and so the response time is longer. In this case, an error was found. The other reason could be a network overload that was not caused by the test. In this case, the fuzzer could not reveal an error. So, the result of this case is: **Undecidable**.

Systemload

As described earlier, the fuzzer can also monitor the CPU usage and the memory load during a test. For these metrics, the retesting can be done similarly to the response time before.

The four retesting cases are:

- **Valid test and retest: normal** - In this case, probably another process occupied the CPU or the memory, when the first test was done. No error is indicated. Therefore the result is: **No Error**.
- **Valid test: normal, retest: abnormal** - In this case, it is very likely that the test itself is the reason for the unusual CPU usage or memory load, because the valid test has the same behavior as in the learning phase. So an abnormal behavior

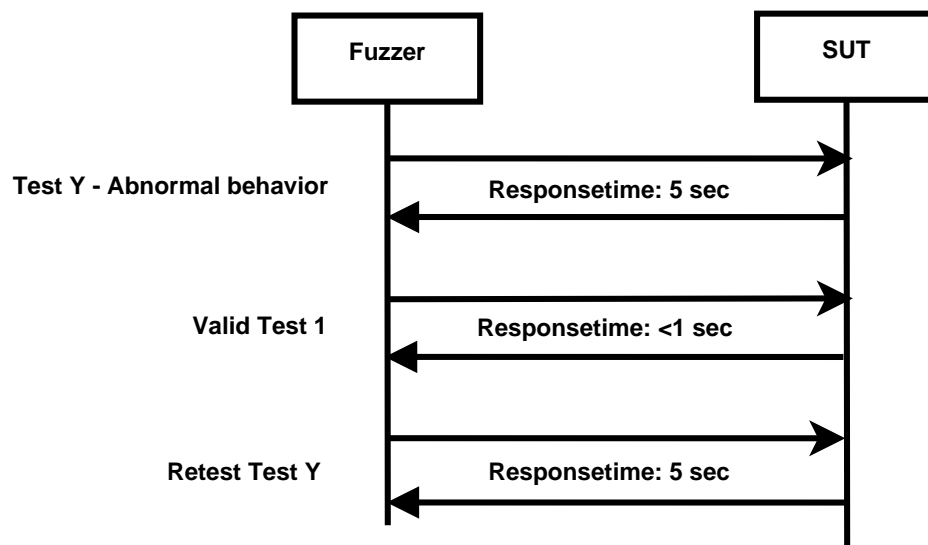


Figure 5.4: Test Y (abnormal behavior) - valid test (normal behavior) - retest test Y (abnormal behavior)

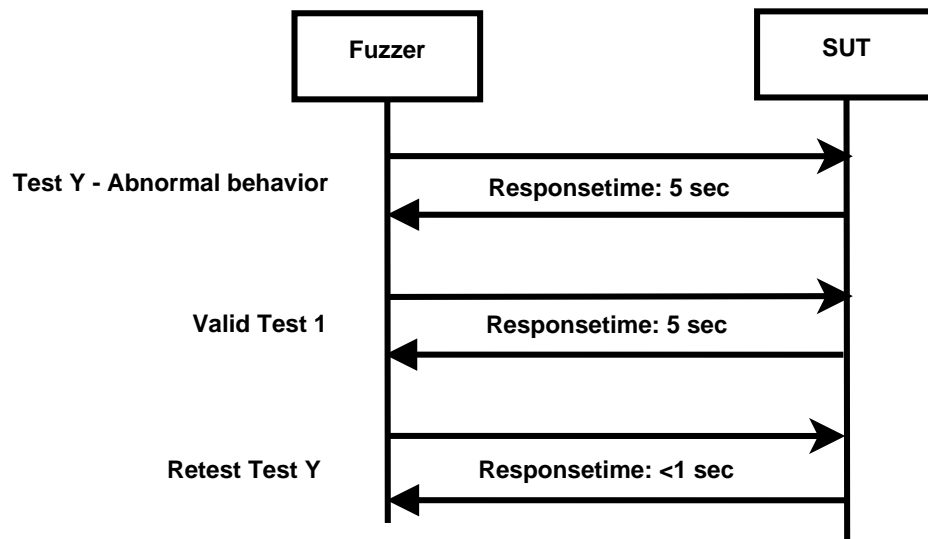


Figure 5.5: Test Y (abnormal behavior) - valid test (abnormal behavior) - retest test Y (normal behavior)

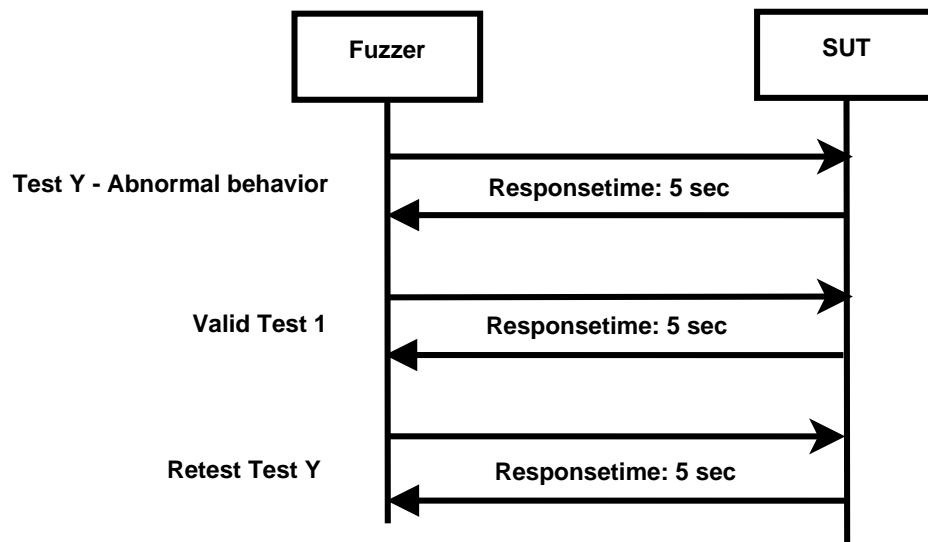


Figure 5.6: Test Y (abnormal behavior) - valid test (abnormal behavior) - retest test Y (abnormal behavior)

was monitored by the fuzzer that could indicate a found error. So the result of this case is: **Error**.

- **Valid test: abnormal, retest: normal** - During the first test and the valid test a high CPU usage or memory load was monitored, but not during the retest. That could indicate that another process was the reason for the higher systemload, because if the test were the reason, then the higher load could also be inspected during the retest. The result of this case is: **No Error**.
- **Valid test and retest: abnormal** - Like the equivalent case of the response time metric, there could be two cases for this behavior: Either the test is the reason for the unusual systemload or another process. So, the result of this case is: **Undecidable**.

GUI of the SUT

When inspecting the GUI, the Analyzer monitors the number of open windows on the SUT. Additional windows like pop-up windows could indicate an error.

Also in this case, the interpretation of the retesting results is similar to the ones of the response time and the system load.

The four retesting cases when monitoring the number of windows in the GUI are:

- **Valid test and retest: normal** - In this case, probably another process has opened a window when the first test was done. No error is indicated, because the number of windows is as expected, when the retest is done. Therefore the result is: **No Error**.
- **Valid test: normal, retest: abnormal** - In this case, it is very likely that the test itself is the reason why the number of windows is unusual (either windows were opened or closed), because the valid test has the same number as in the learning phase. This abnormal behavior could indicate a found error. So the result of this case is: **Error**.
- **Valid test: abnormal, retest: normal** - During the first test and the valid test an unusual number of opened windows was monitored, but not during the retest. That could indicate that another process has opened or closed a windows during the test and the valid test and closed or reopened it during the retest. The result of this case is: **No Error**.
- **Valid test and retest: abnormal** - In this case, the number of open windows has changed for all the tests. Either the first test indicated an unusual opening or closing of windows, what would indicate a found error, or the unusual behavior originated from windows of another program. So, the result of this case is: **Undecidable**.

TCP/UDP Reachability

When testing the reachability, the fuzzer sends a TCP or UDP message to the SUT after each test and waits for a response to check if the system is still available. As explained in subsection 3.4.5, most currently available fuzzers automatically assume that a security error has been detected, if the system has crashed and therefore is not reachable any more.

The four retesting cases when testing if the SUT is still reachable are:

- **Valid test and retest: normal** - In this case, two possible reasons for this behavior exist: In one case the first test caused a crash based on the state the system was in, when the test started. After a restart of the system, the valid test and the retest did not cause a crash. The second case is that the system was not reachable after the first test because of a network problem. Therefore the result is: **Undecidable**.
- **Valid test: normal, retest: abnormal** - In this case, it is very likely that the test itself is the reason that the SUT is not reachable. After the crash caused by the first test, the system restarted for the valid test and then it crashed again when the retest was done. So the result of this case is: **Error**.

- **Valid test: abnormal, retest: normal** - During the first test and the valid test the system was not reachable. In this case, also two possible reasons for this behavior exist: One reason is that the system crashed at the first test because of the state the system was in. Furthermore the system was not restarted until the retest. The other reason could again be a problem of the network that existed during the first test and the valid test. The result of this case is: **Undecidable**.
- **Valid test and retest: abnormal** - In this case also two possible reasons for this behavior exist: One reason is, the system has crashed during the first test and did not restart for the valid test or the retest. The other reason could be a problem with the network. So, the result of this case is: **Undecidable**.

Response message and logfile

At each test case, the fuzzer also collects text data that represent the behavior of the SUT like the response message or the changes in the logfile. In difference to the other metrics discussed in the sections before, the comparison of number values (like the number of open windows or the response time) is not sufficient.

To distinguish differences in the text of valid tests to the text of malicious tests, for example concepts of Text Mining are needed. An example of abnormal behavior in a logfile could be found keywords like *segmentation fault* or *error* that did not occur in the logfile during the tests of the learning phase. Another method is for example a simple comparison of the length of the captured text. If the text is longer than all the texts found in the learning phase this could also indicate a vulnerability like a found SQL injection.

The four retesting cases, when text represents the behavior of the SUT, are:

- **Valid test and retest: normal** - In this case the text of the first test and the retest differ whereas the valid test case behaves normally. This behavior could be caused if threads that run in parallel are involved in the text execution and there an error (e.g. race conditions) occurred that caused the different text results. Therefore the result is: **Error**.
- **Valid test: normal, retest: abnormal** - In this case, it is very likely that the test itself is the reason for the abnormal behavior in the text. The first test and the retest show abnormal behavior (like new keywords), whereas the valid test has the same result as in the learning phase. So the result of this case is: **Error**.
- **Valid test: abnormal, retest: normal** - Like in the first case, the reason for this behavior could be the result of a fault implementation of thread handling. Another possible scenario could be that a crash (caused by the test before) lead to a restart of the system that was completed when the retest was done. So the result of this case is: **Undecidable**.

- **Valid test and retest: abnormal** - In this case also two possible reasons for this behavior exist: One reason is, the system is in an erroneous state caused by the first test that also influenced the result of the valid test. The other reason is that the reason for this behavior is not caused by the test like for example, the logfile is not readable anymore because of changed rights. So, the result of this case is: **Undecidable**.

5.2.3 Vulnerability findings with retesting

The different metrics in combination with retesting can be used to reveal different kinds of vulnerabilities.

An example of possible errors that can be revealed can be seen in the following list:

- **Response time**
 - **DOS** - The DOS could lead to lower performance of the SUT that causes longer response times.
 - **SQL Injection** - In SQL, it is possible to create a delay in the SELECT statement, which also causes longer response times.
- **Systemload**
 - **DOS** - A successful DOS attack on the SUT could lead to a higher memory usage or CPU load.
- **GUI of the SUT**
 - **XSS** - In JavaScript it is possible, to open new windows with the *alert* function. These additional open window could be monitored by the fuzzer.
- **Reachability**
 - **DOS** - If the availability of the SUT is compromised, then it is not reachable any more.
- **Response message and logfile**
 - **Faulty Thread Handling** - A difference in the test result and the result of the retest could indicate a problem with the thread implementation of the application that could cause for example race conditions.

Chapter 6

Proof of concept and evaluation of the retesting concept

In this chapter it is explained, how the concept for retesting together with the other concepts needed for retesting were implemented in the fuzzing framework described before. Furthermore this new version of the fuzzing framework is evaluated by testing an example application with a hidden vulnerability. It is shown that retesting actually helps to reveal security vulnerabilities.

6.1 Proof of concept implementation in the fuzzing framework

All the concepts needed for retesting and the retesting concepts itself have to be implemented in the fuzzing framework.

6.1.1 Monitoring the SUT and indications of security errors

For monitoring the SUT the Analyzers were used that already existed in the fuzzing framework.

6.1.2 Automatic learning, value comparison and Text Mining

For automatic learning of the normal behavior of the SUT, the fuzzer at first needs valid test cases to create a normal behavior on the SUT that can be monitored by the Analyzers. After the learning phase, the fuzzing phase starts, where the fuzzer sends the malicious input values to the SUT. After each of these tests, the fuzzer has to decide, how similar the monitored behavior to the learned behavior is. This can be done with the help of value comparison or Text Mining.

Valid test cases

For the creation of the valid test cases, the *Formal Method* approach was chosen. This is the approach where the tester has to spend less time and effort to obtain the valid test cases than in the *Monitoring* approach because no additional test run is needed to collect valid data. Therefore the *Formal Method* approach is more efficient for the tester. Furthermore with this approach the tester also has more control over the created and used valid cases than with the approach of *collecting typical input data*.

Three different possibilities exist in the fuzzing framework of how the tester can specify the valid test case generation:

- **Random char generation** - The fuzzer can randomly generate char values that can be used as input values for tests. The tester can specify how many of these generated values are needed.
- **Random integer generation** - Random integer values can be generated. The tester can specify what the minimum and maximum values of the generated integers can be and how many of these generated values are needed.
- **Regular expression** - Random strings can be generated that satisfy a regular expression specified by the tester.

Learning Phase

In the learning phase the number of valid tests specified by the tester are sent to the SUT. After each of these tests, the configured Analyzers monitor the behavior.

The Analyzers learn different data during the learning phase:

- **ResponseTimeAnalyzer** - The ResponseTimeAnalyzer monitors the response time and learns the minimum and maximum response time of all requests sent to the SUT.
- **SystemloadAnalyzer** - The SystemloadAnalyzer stores the minimum and maximum CPU load and memory usage of the SUT.
- **GUIAnalyzer** - The GUIAnalyzer stores the minimum and maximum number of open windows during the tests.
- **ResponseAnalyzer** - The ResponseAnalyzer monitors the response messages of the tests. It analyzes each of the texts with the help of Text Mining and categorizes the messages. At the end of the learning phase the ResponseAnalyzer has a list of all different kinds of response messages that occurred during the learning phase.

- **LogfileAnalyzer** - The LogfileAnalyzer monitors the changes in the logfile of the tested application during each test. Like the ResponseAnalyzer, it uses Text Mining and categorizes the logfile entries so that in the end of the learning phase a list of all different kinds of entries exist.

Fuzzing Phase

For the fuzzing phase the tester can specify different files that contain attack vectors. These attack vectors are categorized, based on which kind of vulnerability it could reveal. An example of a category would be: SQL injection. During the fuzzing phase the fuzzer takes one malicious input after the other and sends it to the SUT.

For each of these tests the Analyzers monitor the behavior of the system and compare the values to the data learned in the learning phase.

For example the ResponseTimeAnalyzer (that will also be used for the proof of concept in this thesis) monitors the response time and compares the time to the minimum and maximum response time learned in the learning phase. If the response time is higher or less than these values then this is considered an abnormal behavior and a retesting is be done.

After each test the Analyzers report their result of how likely it is that an error was found to the AnalyzeManager. The Manager takes all these results to calculate an overall result for the current test.

Text Mining

Text mining is needed for all Analyzers that monitor text (like the ResponseAnalyzer or the LogfileAnalyzer). It has to be decided how similar the text is to the text monitored in the learning phase.

The following concepts of Text Mining were implemented:

- **Background knowledge** - Before the testrun the tester can specify a file where keywords are defined that could indicate an error for the tested application.
- **Keyword scores** - For each of the keywords the tester also has to specify a number value that indicates how likely it is that this keyword indicates an error.
- **Stem words** - For all of the keywords defined in the file the fuzzer tries to create new keywords.
- **Case folding** - Before the Analyzer starts searching for keywords in the text, case folding is done.

- **Document sorting** - Document sorting is used to calculate the overall result the Analyzer has to report to the AnalyzeManager.

So for each text the Analyzer looks for the specified keywords. In the fuzzing phase, the Analyzer checks, if a text with the same keywords was already found in the learning phase. If this is the case, then the text is classified as harmless. Otherwise the Analyzer calculates a result based on the keywords and the scores for each keywords. This result is reported to the AnalyzeManager.

6.1.3 Retesting

For the proof of concept in this thesis the retesting procedure of the ResponseTimeAnalyzer will be used. After each test in the fuzzing phase, the ResponseTimeAnalyzer compares the monitored response value to the minimum and maximum response time monitored in the learning phase. If divergences are found, the Analyzer reports to the AnalyzeManager that a retesting is needed. The AnalyzerManager reports to the Fuzzer-Runner that the current test needs retesting. The fuzzer runner then runs the first test of the learning phase and then retests the current test.

After the retest the Analyzer has to decide how likely it is that a vulnerability was found. If based on the result of the retesting it can be assumed that no error has occurred, the Analyzer reports the value *0.0* to the AnalyzeManager. For an error it reports *1.0* and for an undecidable case *0.5*.

6.2 Results using the retesting concept for a sample application

To evaluate the proof of concept implementation an example application containing a SQL injection vulnerability is tested. It is checked if the fuzzer is able to reveal this security problem when monitoring the response time behavior and using the retesting prototype described in the section before.

6.2.1 Retesting for detecting SQL injection vulnerabilities

Functions causing a delay in a SQL statement can be used to slow down the execution of the statement and lead to a higher response time. These timing delays are a commonly used method to reveal SQL injections. [Cla12]

Different SQL servers have implemented different kinds of delay functions:

- **Microsoft SQL Server** - *WAITFOR DELAY 'hours:minutes:seconds'* - The execution of the SQL statement takes the defined timespan longer than it would take normally. [Cla12] [She12a]
- **MySQL Server** - two possibilities: *BENCHMARK* and *SLEEP* - The benchmark function takes two parameters: a function that is executed and a number that tells how often the function should be executed. Using a time consuming function like the calculation of a MD5 hash and a high execution number like 10000000 will cause a delay. The sleep function gets the number of seconds the execution of the statement should take longer as a parameter. [Bel07] [Cla12] [She12a]
- **Oracle** - *DBMS_LOCK.SLEEP* - Oracle server have the function *DBMS_LOCK.SLEEP* that takes a number of seconds as parameter to specify the number of seconds the pause should last. [Cla12] [She12a]
- **PostgreSQL** - *PG_SLEEP* - When using the *PG_SLEEP* function also the number of seconds to sleep can be specified. [Cla12] [RK11]

6.2.2 Description of the simulation application

The SUT is a simulator of a SOAP-webservice with a hidden SQL injection vulnerability. This webservice has the following specification:

- Programming language: PHP
- Webserver: Apache 2.2.14 (Ubuntu)
- Database: MySQL server 5.1.69-0ubuntu0.10.04.1

The reason for using a simulator instead of a real application are as follows: The goal of this testing is not to test a real application but to evaluate the fuzzing framework and to give a proof of concept that the implemented concepts help to reveal vulnerabilities and decrease the number of false positives. When the simulator is used, it is known, which vulnerability exists and it can be checked if this vulnerability can be found. In a real application, the vulnerabilities are not known and therefore also the false-negatives produced by the fuzzing framework are not known. Furthermore the simulator can also be used to simulate a delay in the response (also see version 3 of the simulator).

For simulating different behavior of the simulator and to validate the retesting implementation, the sample application exists in three different versions:

- The first version contains a SQL injection vulnerability.
- In the second version the vulnerability is fixed.

- In the third version the vulnerability is also fixed, but randomly there is a delay in the response.

These versions are described in the following.

Version 1: SQL injection vulnerability in the simulator

The webservice has a login function that takes username and password as input and returns the number of found user accounts with these credentials. The parts of the select statement are concatenated to create the whole select statement. This part of the source code can be seen in listing 6.1.

To exploit the SQL injection vulnerability and to create a delay in the SQL statement execution, an attacker could send as password the input string *' UNION SELECT SLEEP(5) #* to the webservice. The resulting select statement can be seen in listing 6.2. This would create a delay of five seconds when executing the statement.

Listing 6.1: SQL statement in the simulator

```
1 $result = mysqli_query($con ,
2 "SELECT COUNT(*) FROM logins WHERE user = '"
3 . $username . "' AND password ='" . $password . "'");
```

Listing 6.2: SQL injection - created SELECT statement

```
1 SELECT COUNT(*) FROM logins
2 WHERE user = '' AND password ='' UNION SELECT SLEEP(5) #'
```

Version 2: Fixed SQL injection vulnerability in the simulator

To fix the SQL injection vulnerability described in the section before a prepared statement has to be used instead of the string concatenation. [Hay07] This can be seen in listing 6.3.

Listing 6.3: Fixed SQL injection vulnerability on the simulator

```
1 $stmt = mysqli_prepare($con ,
2 'SELECT count(*) FROM logins WHERE user = ? AND password = ? ');
3 mysqli_stmt_bind_param($stmt , 'ss ' , $username , $password );
4 mysqli_stmt_execute($stmt );
```

Version 3: Fixed SQL injection vulnerability and random delay in the response

Like in version 2 a prepared statement is used to fix the SQL injection vulnerability. Additionally for 20 percent of the requests to the webservice the response is delayed as

can be seen in listing 6.4. This is used to simulate a higher response time that is not caused by a found vulnerability and should simulate random delays in the simulator or the network during the security test execution.

Listing 6.4: Random delay

```

1 $int_rand = rand(1,5);
2
3 if($int_rand == 5){
4     sleep(5);
5 }

```

6.2.3 Testing scenarios and expected result

These three versions of the simulator now have to be tested with the fuzzer to evaluate the result of the retesting implementation for the different scenarios of the simulation.

Based on the retesting concept for the response time, five different scenarios could occur during the testing:

- Scenario 1: Test: normal response time
- Scenario 2: Test: delay - Valid Test: normal response time - Retest: normal response time
- Scenario 3: Test: delay - Valid Test: normal response time - Retest: delay
- Scenario 4: Test: delay - Valid Test: delay - Retest: normal response time
- Scenario 5: Test: delay - Valid Test: delay - Retest: delay

The expected results are:

- When testing version 1 of the simulator (contains the SQL injection vulnerability), scenario 3 reveals the vulnerability.
- When testing version 2 of the simulator (where the vulnerability is fixed), then only normal behavior is monitored because there is no SQL injection possible that could create a delay.
- The other three scenarios occur during testing of version 3 because that version simulates a random delay of the response of the webservice.

6.2.4 Testing specification

The simulator is a webservice that implements a login function. The password input value was chosen to be tested. That means that the malicious input values are used as input for the password field.

The structure of the messages, the fuzzer sends to the simulator, can be seen in listing 6.5. The value for the username is always *user1* whereas the value for the password is the placeholder *@@id@@*. For each test, this placeholder is replaced by a malicious input value and then this generated message is sent to the simulator.

Listing 6.5: Message sent by the fuzzer to the simulator

```

1 <?xml version="1.0"?>
2 <soapenv:Envelope>
3 <soapenv:Body>
4     <ns1:login>
5         <username xsi:type="xsd:string">user1</username>
6         <password xsi:type="xsd:string">@@id@@</password>
7     </ns1:login>
8 </soapenv:Body>
9 </soapenv:Envelope>

```

Learning phase

In the learning phase harmless input values are sent to the simulator.

The following valid values were chosen:

- The string *password1* - The login credentials *user1* (can be seen in listing 6.5) and *password1* are valid input values the webservice accepts and that enable a user to login at the webservice. This input data was chosen, so that the fuzzer learns, how the application behaves when valid login data is used.
- Randomly generated integer values - Randomly generated integer values are sent to the fuzzer as password value. So the fuzzer learns, how the application behaves when an invalid but harmless value is sent to the simulator.
- Randomly generated characters - Randomly generated char values are sent to the webservice as a password. These are not the right passwords for *user1* (can be seen in listing 6.5) and therefore the fuzzer learns how the simulator behaves when an invalid but harmless char is sent to the login function.

Fuzzing phase

In the fuzzing phase 2369 different malicious input values are used to generate tests. These input values are attack vectors that are known to possibly exploit vulnerabilities,

Table 6.1: Results of the evaluation of the prototype using a simulation

	Scenario1	Scenario2	Scenario3	Scenario4	Scenario5
Version 1: SQL vuln.	1985	208	172	3	1
Version 2: fixed SQL vuln.	2149	82	138	0	0
Version 3: random delay	1697	298	237	79	58

like for example SQL injection-, XSS-, directory traversal-attacks. All these generated tests are sent to the webservice one after the other. For all of these tests the response time is monitored and if necessary, a retesting is done automatically.

6.2.5 Results of the evaluation

All three versions of the simulator were tested separately. First there was the learning phase with the valid cases and then the fuzzing phase with the 2369 tests. In table 6.1 the results of each version for each scenario can be seen. The results for each of these versions are explained in detail in the following sections.

Results of testing version 1 of the simulator (contains SQL vulnerability)

This version of the simulator contains a SQL vulnerability. Goal of this test is to reveal the vulnerability with the help of the fuzzer. The testing results for version 1 of the simulator can be seen in table 6.1.

- **Scenario 1** - 1985 out of 2369 tests showed normal behavior (scenario 1). In the other 384 cases a retesting had to be done because of an abnormal response time. **No Error: 1962**
- **Scenario 2** - In 208 out of these 384 cases the valid test and the retest showed a normal response time as learned in the learning phase (scenario 2). This indicated that not the test itself was the reason for the delay, because otherwise this delay would also occur during the retest. Without the retesting, these 208 test cases would have been classified as potential found vulnerability. So retesting helped to prevent 208 false positives. **No Error: 208**
- **Scenario 3** - In 172 cases the valid case showed normal, whereas the retests showed abnormal behavior. In 6 of these 172 cases, the sent input was meant to reveal SQL injection vulnerabilities. These were all true positives that revealed the SQL injection vulnerability hidden in the webservice. In 166 cases, the request was blocked by the SOAP server, because the input values were the reason that the messages sent to the SOAP server were malformed. The SOAP sent the answer: *Bad Request*. These false positives could have been avoided by a better learning

phase that also uses input values that provoke a *Bad Request* message. **SQL injection error found by: 6, Malformed SOAP message: 166**

- **Scenario 4** - In three cases the valid test case showed abnormal behavior whereas the retest monitored normal response time. These three cases are input values to exploit XSS attacks. Without the retesting these test cases would have also been classified as found vulnerability. This would have lead to three false positives. **No Error: 3**
- **Scenario 5** - In one retesting case also the valid case and the retesting case showed abnormal behavior. After checking this case manually, it was found out that this was an input to reveal XSS attacks. Therefore this case can be classified as false positive. **Had to be checked manually: 1**

In this proof of concept demonstration the testing results show that retesting helps to reveal vulnerabilities. Furthermore 211 (scenario 2 and scenario 4) false positives were prevented with the help of retesting.

Results of testing version 2 of the simulator (fixed SQL vulnerability)

In this version of the simulator the SQL vulnerability is fixed. When testing this version, no tests should indicate a found SQL injection. The testing results for version 2 of the simulator can also be seen in table 6.1.

- **Scenario 1** - 2149 out of 2369 tests showed normal behavior (scenario 1). In the other 220 cases a retesting had to be done because of an abnormal response time. **No Error: 2146**
- **Scenario 2** - In 82 out of these 220 cases the valid test and the retest showed a normal response time as learned in the learning phase (scenario 2). This indicated that not the test itself was the reason for the delay, because otherwise this delay would also occur during the retest. Without the retesting, these 82 test cases would have been classified as potential found vulnerability. So, retesting helped to prevent 82 false positives. **No Error: 82**
- **Scenario 3** - In 138 cases the valid case showed normal, whereas the retest showed abnormal behavior. None of these cases was caused by an input that should exploit SQL injections. All 138 were again (like in version 1) blocked by the SOAP sever itself with a *Bad Request* message. **False positive SQL injection: 0, Malformed SOAP message: 138**
- **Scenario 4** - No case existed where the valid test case showed abnormal behavior whereas the retest monitored normal response time. **No Error: 0**

- **Scenario 5** - No retesting case was monitored where the valid case and the retesting case showed abnormal behavior. **Had to be checked manually: 0**

No SQL injection error was indicated in the test. Furthermore 82 (scenario 2) false positives were prevented with the help of retesting.

Results of testing version 3 of the simulator (fixed SQL vulnerability and random delay)

In the testing results of version 1 and version 2 it can be seen that scenario 4 and scenario 5 hardly ever occur. Scenario 4 is created, when another circumstance (that is not caused by the test itself) causes a delay in the response of the valid test. In scenario 5 it is undecidable, if the reason for the delay in the valid test and the retest lies in the test itself or other circumstances.

Therefore version 3 of the simulator was created to simulate scenario 4 and scenario 5. This is done by randomly delaying 20 percent of the tests. It has to be ensured that no delay occurs during the learning phase, because otherwise the results are similar to those of version 2. The testing results for version 3 of the simulator can be seen in table 6.1.

- **Scenario 1** - 1697 out of 2369 tests showed normal behavior (scenario 1) and were not affected by the random delay. In the other 672 cases a retesting was necessary because of monitored abnormal behavior. **No Error: 1697**
- **Scenario 2** - In 298 out of this 672 cases the valid test and the retest showed a normal response time as learned in the learning phase (scenario 2). So not the test itself was the reason for the delay, because otherwise the delay would also occur during the retest. Without the retesting, these 298 test cases would have been classified as potential found vulnerability. Retesting helped to prevent 298 false positives. **No Error: 298**
- **Scenario 3** - In 237 cases the valid case showed normal, whereas the retest showed abnormal behavior. In 158 cases the reason for the delay was the malformed SOAP message where the SOAP server sent the response *Bad Request*. In the other 98 cases, the delay during the retest was caused by the delay simulation in the simulator. **SQL injection error found by: 0, Malformed SOAP message: 158, Other false positives: 98**
- **Scenario 4** - In 79 cases the valid test case showed abnormal behavior whereas the retest monitored normal response time. The normal response time during the retest shows that the delay during the test and the valid test were all due to the simulated delay in the simulator. Without the retesting, these tests would have been classified as false positives because of the delay in the test. **No Error: 79**

- **Scenario 5** - In 58 retesting cases, also the valid case and the retesting case showed abnormal behavior. Scenario 5 it is undecidable, if an error was found or not and therefore these test cases would have to be checked manually. This is not necessary in this case, because these results were created with the help of the simulated delay. Therefore all these cases are false positives. **Undecidable: 58**

With retesting 377 (scenario 2 and scenario 4) false positives were prevented.

Discussion of the testing results

The testing of version 1 of the simulator has shown that it is possible to reveal vulnerabilities with the help of retesting: during the test itself a delay was monitored, the valid test behaved normally and the retest also had the delay in the response. Without the retesting it could have also be assumed that the reason for the delay was not the test itself but other circumstances. The retest helps reassuring that the delay was not a random behavior but the reason lies in the test itself.

Furthermore the retesting helped to prevent false positives in all versions of the simulator every time scenario 2 or scenario 4 occurred. The retesting revealed that in these cases the delay was not caused by the test but was due to other reasons like a high network load.

The remaining false positives (in scenario 3), due to the *Bad Request* error, could have been prevented with the help of an improved learning phase.

Summary of the testing results

The hidden SQL vulnerability could be revealed with the help of retesting. In this case, all false positives caused by random delays in the response time could be prevented and about 56 percent of all false positives. When testing the simulator with the fixed vulnerability, no found SQL injection vulnerability was reported by the fuzzer. In this case again, all false positives caused by random delays could be prevented and about 37 percent of all false positives. In these two cases, the other false positives can be prevented with additions valid tests in the learning phase.

When testing the simulator with the randomly created delay of the response time, then about 56 percent of the false positives were prevented.

Chapter 7

Conclusion

Quality assurance is important in the the software development process, because defects can lead for example to high economic damage for companies and their customers. [Dic11] One part of quality assurance is testing for security errors. The goal of security testing is finding as many vulnerabilities as possible.

Security errors are unexpected side effects of an application. [Tho03] Therefore there are no requirements that can be tested with the help of functional test but other methods have to be applied to reveal this unwanted behavior.

One way to do so, is the usage of automatic security testing tools like so called fuzzers. Most fuzzers use the fact that the tested application has crashed or not as only evidence to decide if a security error has been found. This approach is not sufficient to find more complex vulnerabilities like SQL injections or XSS attacks.

In chapter 2 a short introduction to software security was given. The importance of software security and the definition of security were explained. In chapter 3 the whole software testing process and the security development process were introduced. Furthermore the advantages of test automation and the different kinds of test automation were discussed. One kind of automated security testing is explained in detail: fuzzing. A fuzzer is a tool that sends malicious input values to the SUT and monitors the behavior of the system. Based on the inspected behavior it tries to decide if a security error could be found or not.

As part of this thesis an already existing fuzzing framework, described in chapter 4, was used as basis for the self elaborated retesting concept implemented into this framework. For elaborating the retesting concept, different basic concepts were needed like automatic learning as it is already used in intrusion detection systems or Text Mining. The concepts of monitoring the SUT, automatic learning of the normal behavior of the SUT and Text Mining were also described in a paper where the fuzzing framework is used to reveal vulnerabilities in VoIP softphones. [STP⁺11] All these concepts were also examined in chapter 4.

The retesting concept was introduced in chapter 5. The fuzzer first learns in a learning phase the normal behavior of the SUT by sending valid tests to the application. Examples for behaviors that can be monitored would be the response message, the response time or the CPU usage. After the learning phase, the testing phase starts where malicious input values are sent to the SUT. If a behavior deviating from the learning phase is inspected, the retesting has to be done. Therefore a valid test is sent to the SUT and then the malicious input value is sent again. For each of these three tests (test, valid test and retest) the behavior is monitored. Based on these values it can be decided, if an actual error was found or if the results were influenced by other circumstances that were not caused by the malicious input. An example would be a delay in the response time that was caused by a higher network load than in the learning phase.

In chapter 6 it was described how the concepts introduced in the chapters before were implemented in the fuzzing framework. Furthermore a sample application with a hidden SQL injection was used for the evaluation of the fuzzing framework. The evaluation showed that the fuzzer was able to reveal the hidden vulnerability and it was possible to reduce the number of false positives.

The goal of this thesis was to improve the error detection rate during automated security tests by introducing a retesting concept. This goal could be reached: As the evaluation has shown, the improvement of the fuzzer was successful, because it was possible to reveal the hidden SQL injection, and the retesting contributed to this result as it was the idea in the beginning of this thesis. Not only is it possible to reveal attacks like SQL injections, but also a reduction of the false positive rate was possible: All false positives caused by a random delay could be prevented. The test have also shown, that the other false positives can be prevented with the help of additional tests in the learning phase. The reduction of the false positive rate helps the tester and user of this fuzzing framework because less tests have to be checked manually.

List of Abbreviations

BSIMM	Building Security in Maturity Model
CIA	Confidentiality Integrity Availability
CLASP	Comprehensive Lightweight Application Security Process
CPU	Central processing unit
CSV	Categorization status value
DDOS	Distributed Denial of Service
DOS	Denial of Service
GUI	Graphical User Interface
IDS	Intrusion Detection System
SAMM	Software Assurance Maturity Model
SDL	Security Development Lifecycle
SIP	Session Initiation Protocol
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSDL	Secure Software Development Lifecycle
SUT	System under test
URL	Uniform Resource Locator
VoIP	Voice over IP
XML	Extensible Markup Language
XSS	Cross-Site-Scripting

List of Tables

6.1	Results of the evaluation of the prototype using a simulation	80
-----	---	----

List of Figures

2.1	CIA-triangle [And11]	9
3.1	Steps of a common testing process	13
3.2	One example for the security development lifecycle: 7 touchpoints [McG06]	17
3.3	The costs to fix an error in each phase of the software development lifecycle. [McG06]	18
3.4	Example of interfaces where fuzzing can be useful [TDM08]	24
3.5	Parts of a fuzzer [TDM08]	26
3.6	Example test run of a fuzzer [TDM08]	28
3.7	Example of a SQL injection [Cla12]	29
3.8	SQL query after a successful SQL injection [Cla12]	30
3.9	Example code that is vulnerable to XSS [Fla11]	31
3.10	Example for a harmless URL [Fla11]	31
3.11	Example for a URL that exploits a XSS vulnerability [Fla11]	31
3.12	Example for an URL that exploits a XSS vulnerability and load external code [Fla11]	32
4.1	Functional architecture for a retesting based security error detection	34
4.2	Infrastructure of the fuzzing framework [STP ⁺ 11]	34
4.3	Deviation of specified behavior and actual implemented behavior including unwanted side effects [Tho03]	38
4.4	Logfile entry of an apache server after a child process has crashed [TDM08]	40
4.5	SQL statement sent to a database, monitored by strace [TDM08]	42
4.6	Structure of an anomaly detection system [GLT10]	45
4.7	Decisions made during the fuzzing phase	51
4.8	Graphical representation of clustering [HKP12]	52
4.9	Graphical representation of outliers [HKP12]	53
5.1	Steps for a retesting based security error detection	62
5.2	General concept of retesting	63
5.3	Test Y (abnormal behavior) - valid test (normal behavior) - retest test Y (normal behavior)	66

5.4	Test Y (abnormal behavior) - valid test (normal behavior) - retest test Y (abnormal behavior)	67
5.5	Test Y (abnormal behavior) - valid test (abnormal behavior) - retest test Y (normal behavior)	67
5.6	Test Y (abnormal behavior) - valid test (abnormal behavior) - retest test Y (abnormal behavior)	68

Bibliography

- [AACF12] D. Ameller, C. Ayala, J. Cabot, and X. Franch. How do software architects consider non-functional requirements: An exploratory study. In *20th International Requirements Engineering Conference (RE)*, IEEE, pages 41–50, sept. 2012.
- [AERI10] W.A. Aziz, S.H. El-Ramly, and M.M. Ibrahim. Ip - multimedia sub-system (ims) performance evaluation and benchmarking. In *8th International Conference on Computational Technologies in Electrical and Electronics Engineering (SIBIRCON)*, IEEE, pages 209–214, july 2010.
- [AFWJJMZ11] S. Ai-Fen, T. Wen, H. Jian Jun, and L. Ming Zhu. An effective fuzz input generation method for protocol testing. In *13th International Conference on Communication Technology (ICCT)*, IEEE, pages 728–731, 2011.
- [AG10] S. Aziz and M. Gul. A self learning model for detecting sip malformed message attacks. In *International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*, IEEE, pages 744–749, oct. 2010.
- [AHC10] S. Ao, X. Huang, and O. Castillo. *Intelligent Automation and Computer Engineering*. Springer, 1. edition, 2010.
- [And11] J. Andress. *The basics of information security*. Syngress, 1. edition, 2011.
- [ANNM06] S. Abu-Nimeh, S. Nair, and M. Marchetti. Avoiding denial of service via stress testing. In *International Conference on Computer Systems and Applications*, IEEE, pages 300–307, 2006.
- [Bac10] D. Baca. Identifying security relevant warnings from static code analysis tools through code tainting. In *International Conference on Availability, Reliability, and Security, ARES*, pages 386–390, 2010.
- [BBGM12] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. A taint based approach for smart fuzzing. In *Fifth International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, pages 818–825, 2012.

- [BCR08] A. Bacchelli, P. Ciancarini, and D. Rossi. On the effectiveness of manual and automatic unit test generation. In *Third International Conference on Software Engineering Advances, ICSEA*, pages 252–257, 2008.
- [BD09] A. Barrett and D. Dvorak. A combinatorial test suite generator for gray-box testing. In *Third IEEE International Conference on Space Mission Challenges for Information Technology, SMC-IT*, pages 387 –393, july 2009.
- [Bel07] C. Bell. *Expert MySQL*. Springer, 1. edition, 2007.
- [BGM⁺07] B. Burns, S. Granick, S. Manzuik, P. Guersch, D. Killion, N. Beauchesne, E. Moret, J. Sobrier, M. Lynn, E. Markham, C. Iezzoni, and P. Biondi. *Security Power Tools*. O’Reilly Media Inc., 1. edition, 2007.
- [Bis03] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley, 1. edition, 2003.
- [BJC10] C. Bai, C. Jiang, and K. Cai. A reliability improvement predictive approach to software testing with bayesian method. In *29th Chinese Control Conference (CCC)*, pages 6031–6036, 2010.
- [BK10] M. Berry and J. Kogan. *Text Mining Applications and Theory*. Wiley, 1. edition, 2010.
- [BNS⁺08] D. Brumley, J. Newsome, D. Song, Hao Wang, and S. Jha. Theory and techniques for automatic generation of vulnerability-based signatures. *Transactions on Dependable and Secure Computing, IEEE*, 5(4):224–241, 2008.
- [BP88] B.W. Boehm and P.N. Papaccio. Understanding and controlling software costs. *Transactions on Software Engineering, IEEE*, 14(10):1462 –1477, oct 1988.
- [BSA⁺11] A. Biyani, G. Sharma, J. Aghav, P. Waradpande, P. Savaji, and M. Gautam. Extension of spike for encrypted protocol fuzzing. In *Third International Conference on Multimedia Information Networking and Security (MINES)*, pages 343–347, 2011.
- [CGG⁺09] C. Callegari, R.G. Garroppo, S. Giordano, M. Pagano, and F. Russo. A novel method for detecting attacks towards the sip protocol. In *International Symposium on Performance Evaluation of Computer Telecommunication Systems, SPECTS*, volume 41, pages 268 –273, july 2009.
- [Cla12] J. Clarke. *SQL Injection Attacks and Defense*. Syngress Media, 2. edition, 2012.

- [CWKK09] P.M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *Symposium on Security and Privacy, IEEE*, pages 110–125, 2009.
- [Dan10] F. Dane. *Evaluating Research: Methodology for People Who Need to Read Research*. SAGE Publications, Inc, 1. edition, 2010.
- [Dic11] John B. Dickson. Software security: is ok good enough? In *Proceedings of the first ACM conference on Data and application security and privacy*, CODASPY, pages 25–26, New York, NY, USA, 2011. ACM.
- [DV05] H. Debar and J. Viinikka. *Intrusion Detection: Introduction to Intrusion Detection and Security Information Management*. Springer Berlin Heidelberg, 1. edition, 2005.
- [Fla11] D. Flanagan. *JavaScript The Definitive Guide*. O’Reilly-Verlag, 6. edition, 2011.
- [FS06] R. Feldman and J. Sanger. *The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge University Press, 1. edition, 2006.
- [GBDW⁺07] J. Gregoire, K. Buyens, B. De Win, R. Scandariato, and W. Joosen. On the secure software development process: Clasp and sdl compared. In *Software Engineering for Secure Systems, SESS: ICSE Workshops. Third International Workshop on*, pages 1–1, 2007.
- [Gee10] D. Geer. Are companies actually using secure development life cycles? *Computer*, 43(6):12–16, 2010.
- [Gli93] V. D. Gligor. *A Guide to Understanding Security Testing and Test Documentation in Trusted Systems*. 1. edition, 1993.
- [GLR09] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *31st International Conference on Software Engineering, ICSE, IEEE*, pages 474–484, 2009.
- [GLT10] A. Ghorbani, W. Lu, and M. Tavallaee. *Network Intrusion Detection and Prevention Concepts and Techniques*. Springer, 1. edition, 2010.
- [Gol12] D. Gollmann. Veracity, plausibility, and reputation. In *Information Security Theory and Practice. Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems*, volume 7322 of *Lecture Notes in Computer Science*, pages 20–28. Springer Berlin Heidelberg, 2012.
- [GS09] N. Geopalan and B. Sivaselvan. *Data Mining: Techniques and Trends*. PHI Learning Private Limited, 1. edition, 2009.

- [Gup11] G. K. Gupta. *Introduction to Data Mining with Case Studies*. PHI Learning Private Limited, 2. edition, 2011.
- [GVVEB08] D. Graham, E. Van Veenendaal, I. Evans, and R. Black. *Foundations of Software Testing, ISTQB Certification*. Cengage Learning EMEA, revised edition, 2008.
- [Hay07] H. Hayder. *Object-Oriented Programming with Php5: Learn to leverage PHP5's OOP features to write manageable applications with ease*. Packt Publishing, 1. edition, 2007.
- [HHSB10] J. Hinthbergen, K. Hintzbergen, A. Smulders, and H. Baars. *Foundations of Information Security based on ISO27001 and ISO27002*. Van Haren Publishing, 2. edition, 2010.
- [HKP12] J. Han, M. Kamber, and J. Pei. *Data Mining Concepts and Techniques*. Elsevier Inc., 3. edition, 2012.
- [HO08] M.J. Harrold and A. Orso. Retesting software during development and maintenance. In *Frontiers of Software Maintenance, FoSM*, pages 99–108, 2008.
- [HS12] A. Hazeyama and H. Shimizu. Development of a software security learning environment. In *13th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel Distributed Computing (SNPD), ACIS*, pages 518–523, 2012.
- [Hus04] S. Husbey. *Innocent code: a security wake-up call for Web programmers*. Wiley, 1. edition, 2004.
- [HYDD08] K. Hyounghun, C. Younghun, L. Dohoon, and L. Donghoon. Practical security testing using file fuzzing. In *10th International Conference on Advanced Communication Technology, ICACT*, volume 2, pages 1304–1307, feb. 2008.
- [Jac11] S. Jackson. *Research Methods and Statistics: A Critical Thinking Approach*. Wadsworth Publishing, 4. edition, 2011.
- [JME08] C.C. Juan, J.B. Michael, and C.S. Eagle. Vulnerability analysis of hd photo image viewer applications. In *Second International Conference on Secure System Integration and Reliability Improvement, SSIRI*, pages 1–7, july 2008.
- [JSF06] Z. Jianyin, S. Sen, and Y. Fangchun. Detecting race conditions in web services. In *International Conference on Telecommunications, AICT-ICIW. International Conference on Internet and Web Applications and Services/Advanced*, pages 184–184, 2006.

- [JZ06] Z. Jiong and M. Zulkernine. Anomaly based network intrusion detection with unsupervised outlier detection. In *International Conference on Communications, ICC, IEEE*, volume 5, pages 2388–2393, 2006.
- [KBP02] C. Kaner, J. Bach, and B. Pettichord. *Lessons Learned in Software Testing: a context-driven approach*. Wiley Computer Publishing, 1. edition, 2002.
- [KHK10] T. Kitagawa, M. Hanaoka, and K. Kono. Aspfuzz: A state-aware protocol fuzzer based on application-layer protocols. In *Symposium on Computers and Communications (ISCC), IEEE*, pages 202–208, june 2010.
- [KJ09] N. Khakpour and S. Jalili. Using supervised and transductive learning techniques to extract network attack scenarios. In *14th International CSI Computer Conference, CSICC*, pages 71–76, 2009.
- [KMRV03] C. Kruegel, D. Mutz, W. Robertson, and F. Valeur. Bayesian event classification for intrusion detection. In *Computer Security Applications Conference. Proceedings. 19th Annual*, pages 14–23, 2003.
- [KPP11b] M. Kirmse, U. Petersohn, and E. Paffrath. Optimized test error detection by probabilistic retest recommendation models. In *Test Symposium (ATS), 2011 20th Asian*, pages 317–322, 2011.
- [Lig09] P. Liggesmeyer. *Software-Qualität Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag Heidelberg, Springer, 2. edition, 2009.
- [LJC09] X. Luo, W. Ji, and L. Chao. T3fah: A ttcn-3 based fuzzer with attack heuristics. In *WRI World Congress on Computer Science and Information Engineering*, volume 7, pages 744–749, 2009.
- [LWT⁺08] G. Liu, G. Wu, Z. Tao, J. Shuai, and Z. Tang. Vulnerability analysis for x86 executables using genetic algorithm and fuzzing. In *Third International Conference on Convergence and Hybrid Information Technology, ICCIT*, volume 2, pages 491–497, nov. 2008.
- [MAM11] S. Mohanty, A.A. Acharya, and D.P. Mohapatra. A model based prioritization technique for component based software retesting using uml state chart diagram. In *3rd International Conference on Electronics Computer Technology (ICECT)*, volume 2, pages 364–368, 2011.
- [MCF10] N. Mansour, M. Chehab, and A. Faour. Filtering intrusion detection alarms. *Cluster Computing*, 13(1):19–29, 2010.
- [McG06] G. McGraw. *Software security: building security in*. Pearson Education Inc., 1. edition, 2006.

- [MSM12] T. Mattos, A. Santin, and A. Malucelli. Mitigating xml injection zero-day attack through strategy-based detection system, 2012.
- [Mul09] C. Mulliner. Vulnerability analysis and attacks on nfc-enabled mobile phones. In *International Conference on Availability, Reliability and Security, ARES*, pages 695 –700, march 2009.
- [MVS10] K.K. Mohan, A.K. Verma, and A. Srividya. Software reliability estimation through black box and white box testing at prototype level. In *2nd International Conference on Reliability, Safety and Hazard (ICRESH)*, pages 517 –522, dec. 2010.
- [Mye11] G. Myers. *The Art of Software Testing*. Wiley Computer Publishing, 3. edition, 2011.
- [NBA10] F.J.B. Nunes, A.D. Belchior, and A.B. Albuquerque. Security engineering approach to support software security. In *6th World Congress on Services (SERVICES-1)*, pages 48–55, 2010.
- [OI10] F. Omar and S. Ibrahim. Designing test coverage for grey box analysis. In *10th International Conference on Quality Software (QSIC)*, pages 353–356, 2010.
- [PA10] G. Pannell and H. Ashman. *User Modelling for Exclusion and Anomaly Detection: A Behavioural Intrusion Detection System*. Springer Berlin Heidelberg, 1. edition, 2010.
- [PAR⁺12] H. Prahofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger. Opportunities and challenges of static code analysis of iec 61131-3 programs. In *17th Conference on Emerging Technologies Factory Automation (ETFA), IEEE*, pages 1–8, 2012.
- [RCCP08] R. Rogers, M. Carey, P. Criscuolo, and M. Petruzzi. *Nessus Network Auditing*. Elsevier, Inc., 2. edition, 2008.
- [RK11] S. Riggs and H. Krosing. *PostgreSQL 9 Administration Cookbook: LITE : Configuration, Monitoring and Maintenance*. Packt Publishing, 1. edition, 2011.
- [RMZR10] J. Romero-Mariona, H. Ziv, and D.J. Richardson. Increasing trustworthiness through security testing support. In *Second International Conference on Social Computing (SocialCom), IEEE*, pages 920 –925, aug. 2010.
- [RRT12] R.R. Rejimol Robinson and C. Thomas. Evaluation of mitigation methods for distributed denial of service attacks. In *Conference on Industrial Electronics and Applications (ICIEA), IEEE*, pages 713–718, 2012.

- [Sch00] B. Schneier. *Secrets & Lies*. John Wiley & Sons, 1. edition, 2000.
- [Sch13] C. Schanes. *Scope and Depth Efficient Testing Approach and Framework for Enhancing the Detection of IT Security Bugs*. Dissertation 2013.
- [SGA07] M. Sutton, A. Greene, and P. Amini. *Fuzzing Brute Force Vulnerability Discovery*. Addison-Wesley, 1. edition, 2007.
- [SH10] L. K. Shar and B. K. T. Hee. Auditing the defense against cross site scripting in web applications. In *Proceedings of the 2010 International Conference on Security and Cryptography (SECRYPT)*, pages 1–7, 2010.
- [She12a] M. Shema. *Hacking Web Apps: Detecting and Preventing Web Application Security Problems*. Syngress Media, 1. edition, 2012.
- [SHE12b] T. Sommestad, H. Holm, and M. Ekstedt. Effort estimates for vulnerability discovery projects. In *45th Hawaii International Conference on System Science (HICSS)*, pages 5564–5573, 2012.
- [SHFG13] C. Schanes, A. Hübler, F. Fankhauser, and T. Grechenig. Generic approach for security error detection based on learned system behavior models for automated security tests. In *Sixth International Conference on Software Testing, Verification and Validation (ICST), IEEE*, 2013.
- [SM11] R. Shirvani and N. Modiri. Software architectural considerations for the development of secure software systems. In *7th International Conference on Networked Computing (INC)*, pages 84–88, 2011.
- [SP10] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Symposium on Security and Privacy (SP), IEEE*, pages 305–316, 2010.
- [STP⁺11] C. Schanes, S. Taber, K. Popp, F. Fankhauser, and T. Grechenig. Security test approach for automated detection of vulnerabilities of sip-based voip softphones. In *International Journal on Advances in Security*, volume 4, pages 95–105, 2011.
- [TCZ⁺09] H. Tang, H. Chen, G. Zhao, Q. Liu, and J. Zhao. The vulnerability analysis framework for java bytecode. In *15th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 896–901, dec. 2009.
- [TDB12] P. Tsankov, M.T. Dashti, and D. Basin. Secfuzz: Fuzz-testing security protocols. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 1–7, 2012.
- [TDM08] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., 1. edition, 2008.

- [Tho03] H.H. Thompson. Why security testing is hard. *Security Privacy, IEEE*, 1(4):83–86, 2003.
- [TLJ⁺12] F. Thung, D. Lo, Lingxiao Jiang, Lucia, F. Rahman, and P.T. Devanbu. When would this bug get reported? In *28th International Conference on Software Maintenance (ICSM), IEEE*, pages 420–429, 2012.
- [TSG10] M. Tavallaei, N. Stakhanova, and A.A. Ghorbani. Toward credible evaluation of anomaly-based intrusion-detection methods. *Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE*, 40(5):516–524, 2010.
- [TSH⁺10] S. Taber, C. Schanes, C. Hlauschek, F. Fankhauser, and T. Grechenig. Automated security test approach for sip-based voip softphones. In *The Second International Conference on Advances in System Testing and Validation Lifecycle, August 2010, Nice, France*. IEEE Computer Society Press, August 2010.
- [VK12] D. Vidhate and P. Kulkarni. Cooperative machine learning with information fusion for dynamic decision making in diagnostic applications. In *International Conference on Advances in Mobile Network, Communication and its Applications (MNCAPPS)*, pages 70–74, 2012.
- [Wan04] A.J.A. Wang. Security testing in software engineering courses. In *Frontiers in Education, FIE*, pages F1C – 13–18 Vol. 2, oct. 2004.
- [WFH11] I. Witten, E. Frank, and M. Hall. *Data Mining Practical Machine Learning Tools and Techniques*. Elsevier Inc., 3. edition, 2011.
- [WM10] J. Watkins and S. Mills. *Testing IT: an off-the-shelf software testing process*. Cambridge University Press, 2. edition, 2010.
- [XJ10] W. Xiaojun and S. Jinhua. The study on an intelligent general-purpose automated software testing suite. In *International Conference on Intelligent Computation Technology and Automation (ICICTA)*, volume 3, pages 993–996, 2010.
- [XSJJ11] C. Xiuzhen, L. Shenghong, M. Jin, and L. Jianhua. Quantitative threat assessment of denial of service attacks on service availability. In *International Conference on Computer Science and Automation Engineering (CSAE), IEEE*, volume 1, pages 220–224, 2011.
- [YD12] S. Yu and H. Dake. Model checking for the defense against cross-site scripting attacks. In *International Conference on Computer Science Service System (CSSS)*, pages 2161–2164, 2012.

- [YH12] Y. Yingbing and W. Han. Anomaly intrusion detection based upon data mining techniques and fuzzy logic. In *International Conference on Systems, Man, and Cybernetics (SMC)*, IEEE, pages 514–517, 2012.
- [YMTT05] A. Yamada, Y. Miyake, K. Takemori, and T. Tanaka. Intrusion detection system to detect variant attacks using learning algorithms with automatic generation of training data. In *International Conference on Information Technology: Coding and Computing, ITCC*, volume 1, pages 650–655 Vol. 1, 2005.
- [YT11] Z. Yu and J. Tsai. *Intrusion Detection A Machine Learning Approach*. Imperial College Press, 3. edition, 2011.
- [ZHYS11] T. Zhushou, Z. Haojin, C. Zhenfu, and Z. Shuai. L-wmxd: Lexical based webmail xss discoverer. In *Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, pages 976 –981, april 2011.

Weblinks

- [99] Owasp samm security testing. https://www.owasp.org/index.php/SAMM_-_Security_Testing_-_2. Accessed: 2013-07-09.
- [100] Owasp top 10 2010. https://www.owasp.org/index.php/Top_10_2010-Main. Accessed: 2013-04-05.