FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Automated Detection of Security Vulnerabilites Using Machine Learning for Automated Testing

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Andreas Hübler
Matrikelnummer 0456618

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Christian Schanes

Wien, 6. Oktober 2014

(Unterschrift Verfasser/In)

(Unterschrift Betreuung)

Automated Detection of Security Vulnerabilites Using Machine Learning for Automated Testing

Master's Thesis

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computational Intelligence

by

Andreas Hübler
Registration Number 0456618

elaborate at the
Institut of Computer Aided Automation
Reseach Group for Industrial Software
to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Thomas Grechenig
Assistance: Christian Schanes

Vienna, October 6, 2014

Statement by Author

Andreas Hübler
Heiderosenstraße 57, 4600 Wels

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.

I hereby declare that I am the sole author of this thesis, that I have completely indicated all sources and help used, and that all parts of this work – including tables, maps and figures – if taken from other works or from the internet, whether copied literally or by sense, have been labelled including a citation of the source.

(Place, Date)	(Signature of Author)

Acknowledgements

I want to thank my advisor Thomas Grechenig for his supervision and the possibility to write my masters thesis at the INSO research group.

My particular thanks goes to my assistant advisor Christian Schanes for the possibility to work at this project and his invaluable support during the creation of this work. Without him, all this would not have been possible.

Further, I want to thank Florian Fankhauser. Without his lectures on IT security I would have never discovered the fascination of this topic.

I am very grateful to my family for their unhesitant support during all these past years.

Finally my very special thanks to the love of my life for her incredible help in the writing of this document and that she is always there for me.

Abstract

IT security is an important aspect of the software development life cycle and thorough security testing is necessary to ensure the quality of the software. To cope with the increasing complexity and amount of software, security testing is required to increase its efficiency through the development of automated testing methods. However, the manual analysis of the results produced by an automated test execution can be very cumbersome and extremely time consuming.

The purpose of this thesis is to optimize the analysis phase through the automatic evaluation of test results and detection of suspicious test cases. In this work the optimization task was approached by using methods from the research field of machine learning. To this aim several supervised and unsupervised machine learning methods were investigated: Artificial Neural Network, Self-Organizing Map, Support Vector Machine, c-Means Clustering. Based on the prototypes created in this thesis, the four approaches were evaluated in terms of their applicability in detecting suspicious behaviour of the test subject.

The results show that machine learning methods are well suited for an integration into a security testing framework. Three out of four implementations displayed a high vulnerability detection rate while maintaining a straightforward implementation, fast running times and easy adaptation into various projects. Integrated into a security testing system, machine learning methods can improve the analysis phase of the test execution. With their help the amount of manual labor can be decreased without any loss of vulnerability detection rate.

Keywords

Machine learning; System testing; Optimization; Security

Kurzfassung

IT Sicherheit ist ein wichtiger Aspekt in der Entwicklung von Software Projekten. Um die Qualität der Software zu gewährleisten, ist es notwendig, die Sicherheitsmaßnahmen gründlich zu testen. Um mit der wachsenden Menge und Komplexität von Programmen zurechtzukommen, müssen auch Sicherheitstests ihre Effizienz durch die Entwicklung von automatisierten Testmethoden steigern. Allerdings generieren automatisierte Testmethoden eine Vielzahl an Daten, deren manuelle Auswertung umständlich und zeitaufwändig ist.

Diese Arbeit hat sich das Ziel gesetzt, die Auswertungs- und Analysephase durch die automatische Evaluation der Testresultate zu optimieren und auffällige Testfälle zu erkennen. Als Herangehensweise an diese Zielsetzung wurden Methoden aus dem Forschungsgebiet des maschinellen Lernens gewählt. Es wurden vier Methoden des überwachten und nicht-überwachten maschinellen Lernens eingesetzt: künstliche neuronale Netze, selbstorganisierende Karten, Support Vector Machine sowie k-Means Clustering. In dieser Arbeit wurden Prototypen der jeweiligen Methoden erstellt und evaluiert, wie gut sie verdächtiges Verhalten des zu testenden Systems erkennen.

Die Resultate zeigen, dass die Methoden aus dem Bereich des maschinellen Lernens gut für die Integration in ein bestehendes Sicherheitstest-Programm geeignet sind. Drei der vier getesteten Implementierungen erkannten Schwachstellen mit einer hohen Genauigkeit mit den zusätzlichen Vorteilen einer unkomplizierten Anwendung, schnellen Laufzeiten und der Möglichkeit, sie einfach in verschiedene Software-Projekte einzubauen. Integriert in einem Sicherheitstest-Programm können dieses Methoden die Auswertung von Sicherheitstests verbessern. Mit ihrer Hilfe kann die manuelle Arbeitszeit bei gleichbleibender Erkennungsrate von Schwachstellen deutlich verringert werden.

Schlüsselwörter

Maschinelles Lernen; Systemtest; Optimierung; Sicherheit

Cake, and grief counseling, will be available at the conclusion of the test.

– GLaDOS

Contents

1 Introduction **1**
1.1 Expected Result . . . 1
1.2 Methodological Approach . . . 2
1.3 Organization of this Thesis . . . 2

2 Introduction to IT Security and Testing **4**
2.1 Basics of IT Security . . . 4
2.1.1 Security Needs . . . 4
2.1.2 Definitons in IT Security . . . 6
2.1.3 Types of Attacks . . . 8
2.1.4 Measures against Threats . . . 12
2.2 Software Testing . . . 14
2.2.1 Black Box Testing . . . 15
2.2.2 White Box Testing . . . 16
2.2.3 Gray Box Testing . . . 17
2.3 Testing of IT Security . . . 17
2.3.1 Fuzz Testing . . . 17
2.3.2 Penetration Testing . . . 18
2.4 Challenges of Security Testing . . . 20

3 Artificial Intelligence in IT Security **21**
3.1 Machine Learning . . . 21
3.1.1 Classification . . . 22
3.1.2 Clustering . . . 23
3.2 Artificial Neural Network . . . 24
3.2.1 Biological Model of a Neuron . . . 24
3.2.2 Mathematical Model of a Neuron . . . 24
3.2.3 Neural Network Architecture . . . 28
3.2.4 Learning in Artificial Neural Networks . . . 29
3.3 Self-Organizing Maps . . . 31
3.3.1 Learning in Self-Organizing Maps . . . 32
3.3.2 Example . . . 33
3.4 Support Vector Machines . . . 34
3.5 Clustering . . . 36
3.5.1 Hard c-Means . . . 37
3.5.2 Fuzzy c-Means . . . 38
3.6 State of the Art . . . 40
3.6.1 Artificial Neural Network . . . 40
3.6.2 Self-Organizing Map . . . 40
3.6.3 Support Vector Machine . . . 41
3.6.4 Clustering . . . 41

4 Architecture of the PoC Implementation **43**
4.1 Architecture of Security Testing Frameworks . . . 43

4.2 The Fuzzing Framework ”Fuzzolution” 44
4.2.1 Process of one Fuzzing Test Run 45
4.2.2 Composition of the fuzzolution tool 45
4.3 Integration of the AI System 48
4.3.1 Representation of the System Behavior Model 49
4.3.2 Integration of the Prototype Implementations 50

5 Setup of Test Environment **52**
5.1 Setup of Test Environment 52
5.2 Description of System Under Test 54
5.2.1 WAVSEP 55
5.2.2 ZAPWAVE 56

6 Evaluation of PoC Implementations **58**
6.1 Evaluation Method 58
6.2 Artificial Neural Network (ANN) 58
6.2.1 Design Decisions for the Implementation 59
6.2.2 Results of the Evaluation 61
6.3 Self-Organizing Map (SOM) 61
6.3.1 Design Decisions for the Implementation 61
6.3.2 Results of the Evaluation 64
6.4 Support Vector Machine (SVM) 65
6.4.1 Design Decisions for the Implementation 65
6.4.2 Results of the Evaluation 67
6.5 Clustering Algorithm 67
6.5.1 Design Decisions for the Implementation 67
6.5.2 Results of the Evaluation 69

7 Summary and Discussion of Evaluation Results **70**
7.1 Results of the Evaluation 70
7.1.1 Support Vector Machine 72
7.1.2 Self-Organizing Map 72
7.1.3 Clustering 73
7.1.4 Artificial Neural Network 73
7.2 Discussion of Results 74

8 Conclusion **76**

Bibliography **78**

List of Figures

2.1 The relationship between the main security needs. [64] 5
2.2 A taxonomy of security flaws by Landwehr *et al.* . [45] 9
2.3 An example application with XSS vulnerability. 11
2.4 A picture about threats, vulnerabilities and control. [64] 13
2.5 An illustration of a black box test. [79] 15
2.6 The control flow graph of an example program. [59] 16
2.7 All phases of a complete penetration test. [17] 18
2.8 Intended and implemented functionality in software. [82] 20

3.1 The training data for the classification example. (based on [3]) 22
3.2 A example clustering of input data consisting of three clusters. (based on [3]) 23
3.3 The biological model of a neural cell. [27] 25
3.4 A mathematical model of a neuron. [27] 26
3.5 The transformation produced by adding a bias b_k. [27] 27
3.6 An example of a fully connected single-layer neural network. (based on [27]) 28
3.7 An example of a partially connected multi-layer neural network. (based on [27]) . . . 29
3.8 An example of a single-layer neural network with feedback loops. (based on [27]) . . 30
3.9 A SOM with synaptic connections and the winning neuron. [27] 32
3.10 A graphical interpretation of the Gaussian neighbourhood function. [27] 33
3.11 Stages of a SOM training process: (a) the distribution of the training data; (b) the SOM is initialised with random data; (c) during the training of the SOM; (d) training is finished, the maps now spans over the entire dataset, representing its distribution. [42] 34
3.12 A SVM example showing an optimal separating hyperplane. [3] 35
3.13 A training set with a separating hyperplane in two and three dimensions. [72] 36
3.14 An example of a membership function for fuzzy sets. [14] 37

4.1 Architecture for learning system behaviour and determining security failures. [73] . . 44
4.2 Architecture of the fuzzolution framework. (based on [74]) 46
4.3 Sequence diagram describing the test execution of the fuzzolution tool. 47
4.4 Architecture of the extended fuzzolution framework. (based on [74]) 50

5.1 The OWASP Broken Web Applications Project page. 55
5.2 The WAVSEP web interface displaying some application descriptions. 56
5.3 The ZAPWAVE web interface displaying the application categories. 57

6.1 Artificial Neural Network design of the prototype implementation. 59
6.2 Artificial Neural Network parameter evaluation. 60
6.3 A visualisation of a test run evaluated with a SOM. 62
6.4 Self-Organizing Map parameter evaluation. 63
6.5 An example for an SVM used for novelty detection. 64
6.6 Support Vector Machine parameter evaluation. 66
6.7 An example for an c-means clustering used for outlier detection. 67
6.8 Clustering parameter evaluation. 68

List of Tables

2.1 Amount of stolen user information from online community sites. 7
2.2 An excerpt of vulnerable websites taken from the ”xssed” project website. 12

5.1 Detailed values for each analyzer during one example test case. 53

6.1 Evaluation result of the Artificial Neural Network method. 61
6.2 Evaluation result of the Self-Organizing Map method. 64
6.3 Evaluation result of the Support Vector Machine method. 67
6.4 Evaluation result of the c-Means Clustering method. 69

7.1 All results of the Machine Learning (ML) method evaluation combined. 71
7.2 The result of the running time evaluation of the ML methods. 71
7.3 Summary of all results of the ML method evaluation. 74

List of Listings

2.1 A small example presenting an SQL Injection. 10
2.2 An example Uniform Resource Locator (URL) that can cause a path traversal attack. 10
2.3 Typical code used for testing XSS vulnerabilities. 10

4.1 Some examples of attack vectors. 45
4.2 An attack template for an application of the WAVSEP platform. 45

5.1 An example attack string for an application of the WAVSEP platform. 56
5.2 An example attack string for an application of the ZAPWAVE platform. 57

List of Abbreviations

AI Artificial Intelligence. 1–3, 22, 45, 52, 53

ANN Artificial Neural Network. iii, 2, 42, 53, 60, 63, 73–78, 80

BP Back Propagation. 42

CPU Central Processing Unit. 50, 51, 55, 57

DARPA Defense Advanced Research Projects Agency. 43

DNS Domain Name System. 20

DoS Denial of Service. 55

GA Genetic Algorithm. 42

GUI Graphical User Interface. 50

HTTP Hyper Text Transfer Protocol. 47, 49, 56–58, 80

IDS Intrusion Detection System. 14, 42–44

IEEE Institute of Electrical and Electronics Engineers. 15

IP Internet Protocol. 20

MITM Man-In-The-Middle. 8

ML Machine Learning. 22, 23, 42, 74, 77

OWASP Open Web Application Security Project. 9, 12, 56, 58

RBF Radial Basis Function. 43, 67

RFC Request For Comments. 7

RSVM Robust Support Vector Machine. 43

SIP Session Initiation Protocol. 80

SOM Self-Organizing Map. iii, 2, 33–35, 42, 43, 53, 64–66, 73–78, 80

SQL Structured Query Language. 9, 78

SUT System Under Test. 16, 18, 19, 45, 47–57, 61, 64, 76, 78, 80, 81

SVM Support Vector Machine. iii, iv, 2, 36, 42, 43, 53, 67–70, 73–78, 80

TCP Transmission Control Protocol. 49

UML Unified Modeling Language. 48

URL Uniform Resource Locator. 11, 47, 56, 58, 59

VoIP Voice over Internet Protocol. 80

WAVSEP Web Application Vulnerability Scanner Evaluation Project. 47, 54–57, 59, 60, 80

XSS Cross Site Scripting. 9, 11, 12, 57, 59

ZAP Zed Attack Proxy. 58, 59

ZAPWAVE Zed Attack Proxy - Web Application Vulnerability Examples. 54, 56, 58–60, 80

1 Introduction

Information Technology (IT) has become of great importance to our society. We trust IT systems to keep us safe (e.g., airplanes) or to hold our most private data (e.g., health care systems). We know that all these systems are build by humans and therefore contain flaws. The more complex a system is, the more flaws it will contain. With knowledge of this flaws, some adversaries may misuse the system to take over control or steal the confidential data stored. Therefore a very important part of building IT systems is testing: finding these flaws and correcting them.

Fuzz testing is a technique for IT security and quality assurance to find flaws in software. Typically it is a fast and cost efficient way of testing. The most basic way of fuzz testing, or ”fuzzing” as it is often called, is to send abnormal data to a system in order to crash it. From these crashes conclusions to possible flaws can be drawn. Although this is a very simple approach, experience has shown that many programs are not able to handle unexpected input correctly.

The most important modules of a security testing software are the generation of the attack data and the analysis of the behaviour of the system under test (SUT). The data generation module is responsible for a fast coverage of the possible attack data and getting the SUT into an erroneous state as quickly as possible. The analysis module on the other hand has to detect all unexpected behaviour reliably while reducing the false positive feedback to a minimum.

The topics of security testing and especially fuzz testing have received a lot of scientific interest in the recent past (e.g., [24] [1] [8]). Nevertheless, experience from projects with complex systems to be tested has shown that there is still improvement to be done until fuzz testing tools, often shortened to ”fuzzers”, are the fast and general-purpose, easy-to-use tools they are expected to be.

The results of the studies of the thesis will be used to improve an existing fuzzing software project. This fuzzer is a black-box fuzzer and uses state-of-the-art techniques within its components. It has been successfully used in several projects, including the publication of Schanes *et al.* [74].

1.1 Expected Result

Currently, most fuzz testing tools either give no feedback about the behaviour of the tested system or have a simple check, to test if the system is still responding or not. To improve the general efficiency of fuzz testing and generate a useful feedback for the user, a more sophisticated approach is required. As mentioned above, a set of analyzers will monitor some metrics of the system and generate a feedback from these values. Manually programming the analyzers to return useful values for a particular system can be a very time consuming and inefficient process. A more desired approach would be a set of analyzers that can evaluate the SUT with a minimum of manual interaction.

In the research field of Artificial Intelligence (AI) there exist methods that can automatically learn about the behaviour of a system. The methods can classify the behaviour of a system and, for example, tell whether the system is not acting properly.

The thesis will make use of these AI methods and present an approach that will allow the analyzers to learn about the behaviour of the tested system. During the actual testing phase, the analyzers can use the learned knowledge for a better evaluation of the system and give more useful feedback.

This concept was already proposed in an earlier work of the author: ”Generic Approach for Security Error Detection Based on Learned System Behavior Models for Automated Security Tests” by C. Schanes, A. Hübler, F. Fankhauser and T. Grechenig. The publication appeared in the *Proceedings of the Sixth IEEE International Conference on Software Testing, Verification and Validation* in 2013 [73]. The ideas presented in this published work are used as basis and will be extended further in this thesis.

The improvements on these method will have a huge impact on the efficiency of Fuzz Testing. On the one hand this will reduce the amount of time that is required to manually program and configure the test software to interact with the system to test. On the other hand, by yielding more meaningful results such that less manual analyzing has to be done.

1.2 Methodological Approach

The work will start with research on the current development of the topic ”Computational Intelligence” and the combination of this topic with ”IT Security”. There is a lot of research going on in the recent years, with many new approaches and ideas. Some of them are presented in the chapter 3. *Artificial Intelligence in IT Security*. The goal of this study is to find useful and well working combinations of security projects supported by artificial intelligence and to get some ideas on new possibilities.

With the knowledge of this theoretical study, the work will focus on finding possible algorithms that could be used for security testing. These ideas will then be studied further to evaluate if they actually can improve automated testing and how well they could perform on these tasks.

To gain significant results from the evaluation of the algorithms, it is necessary to have a working implementation that can be used for testing and evaluating. Since the implementation and testing of a fully-featured algorithm is a very time consuming process, a proof-of-concept approach was chosen. For this purpose, already existing implementations and frameworks will be used and adapted. These prototypes are then tested with various parameters to develop their full potential within the testing subject. All algorithms will be tested against the same prepared system under test. This system is specially designed for evaluating security testing tools.

The background architecture of the existing Fuzz Testing project will be redesigned for the integration of the algorithms. Finally, the selected prototype algorithms are implemented as proof of concept and used for test execution. These tests are intended to evaluate the applicability of the algorithms and to analyze possible improvements for the test result. Eventually, conclusions will be drawn from these test results.

1.3 Organization of this Thesis

The thesis is organized in three main parts. The first part includes the chapters 2 and 3 and explains the theoretical background upon which the later chapters are based on. In particular, chapter 2 gives information about the topics of *Security* and *Software Testing*. Chapter 3 covers general knowledge about *Machine Learning* and in detail information about the method: *Artificial Neural Network*, *Self-Organizing Map*, *Support Vector Machine*, and Clustering.

The second part of the thesis includes the chapters 4, 5 and 6. Using the knowledge given in chapters 2 and 3, the chapter 4 combines the data to an architectural overview, of how the field of Artificial Intelligence can be used to improve security tests. The following chapter 5 describes the test environment that was used to evalute the methods. Chapter 6 presents the implementations of the methods given in chapter 3 and the results of the prototype evaluation.

The final part consists of the chapters 7 and 8. A summary is given in chapter 7, followed by a detailed discussion of the results. Some conclusions will be drawn in chapter 8 together with a few ideas of how the gained knowledge can be used and further improved.

2 Introduction to IT Security and Testing

The world is a complex place. We build huge systems that support us in our everyday life. Systems far too complex for one person to understand as a whole. Imagine the Internet: millions of computers, each one a incomprehensibly complex system, connected into one giant system. But we still trust these systems to keep our most private secrets, protect our wealth and keep us save from harm. [75]

The world is an interconnected place. With the ever increasing availability of Internet connections, physical interaction becomes less and less important. On the Internet every two points are adjacent, whether they are across the hall or across the planet. Even while travelling around the world, you can log into a server operating in your country, read your mails and check your account balance. However, if one person can do it, others can too, with potentially ill intentions. [75]

The world is an imperfect place. In theory, we can design systems that are provably secure, but we can't actually build them to work securely in the real world. Mathematics is perfect; reality is subjective; Mathematics is defined; computers are ornery. Mathematics is logical; people are erratic, capricious and barely comprehensible. But we can take precautions: *testing* of a system with the intention of improving its security performance is one possibility and the topic of this thesis. [75]

2.1 Basics of IT Security

When dealing with security and especially IT security, the first thing required is to define what *security* means in the current context. In general, security is about the protection of assets. When going into detail, ”security” becomes a highly overloaded word and its meaning can be quite subjective. Anderson [5] brings an example of a corporation and its employees: for the company, security might mean the ability to monitor all online activities of its employees; to the employees, it might mean to be able to do online activities without being monitored.

2.1.1 Security Needs

Following this example it becomes obvious, that many different goals of security exist. A common separation is into three main security needs: confidentiality, integrity and availability [10][64][75]. However, the interpretations of these aspect may wary, depending on the context in which they arise.

The purpose of IT Security is addressing these three goals. The challenge hereby is finding the right balance between these often conflicting goals. Figure 2.1 visualises the security needs and shows that these three characteristics can be independent, can overlap and even be mutually exclusive. A simple example of a conflicting situation is ensuring the confidentiality of some object by simply preventing everyone from accessing it. Although this system preserves the goal of confidentiality it does not meet the requirements of availability. The result is that such a system can, by definition, not regarded as secure. [10]

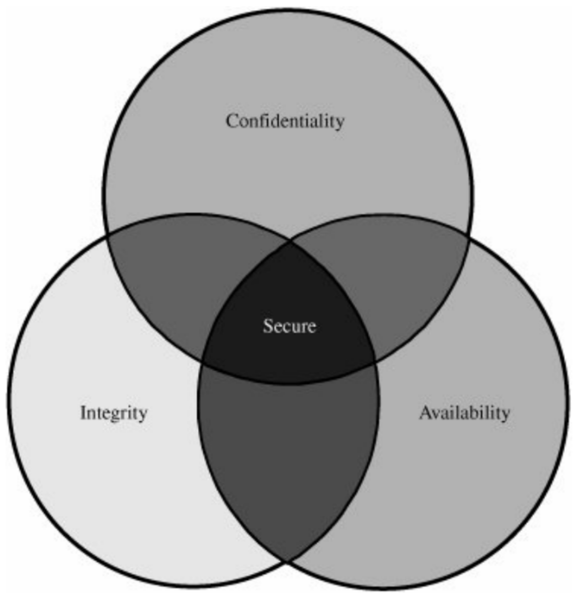


Figure 2.1: The relationship between the main security needs. [64]

Confidentiality The aspect of *confidentiality* ensures that assets are only accessed by parties that are authorized to do so. Meaning that only those individuals who should have access to some resources will actually get access. This is a very general definition, since in this context the term individual can mean several different things, for example a person or a computer program. A resource can be some kind of information like a file, some computer program or the likes. Although this definition seems to be straightforward, ensuring confidentiality can be very difficult to maintain. [64]

Confidentiality is sometimes called *secrecy* or *privacy*. These terms clearly do overlap but then again they are not exactly the same. Secrecy is more of a technical term referring to the effect of the mechanisms used to limit the number of individuals that can access a resource. Cryptography is a typical example for a mechanism ensuring secrecy. Privacy is the ability and right to protect your personal secrets from becoming public knowledge [5]. Hospital patients, for example, have a right to privacy. If some information about their treatment gets public this could lead to embarrassment and to social disadvantages for the patient [75].

Integrity The property of *integrity* deals with the validity of data. It is not concerned with the origin of the data, but whether the data has been modified since its creation. Suppose, a decision about a treatment for a patient is based on the information in the medical record. Then it is very important that the data has not been modified since its collection and still represents the actual knowledge of the condition of the patient. Otherwise a completely different decision could be made, with unknown and probably negative effects. [75]

In a different context the above definition of integrity may differ slightly. The example above talks about an item being unmodified. However, integrity could also refer to a different situation where an item is for example ”modified only in acceptable ways” or ”modified only by authorized people”. Even the interpretation ”meaningful and consistent” may refer to integrity. [64]

Availability *Availability* describes the ability to use a desired resource or information. It is an important aspect of system design, since an unavailable system is equally useful to an non existing system. In a typical design, several mechanisms that are based on statistical models will try to ensure the availability of the system. However, someone may be able to use the system in such a way that the assumptions of the model are no longer valid. This can lead to a situation where the availability can not be maintained any longer. These attempts to block availability are called *denial of service*. [10]

As before, this definition is not generally valid. Different people may expect different things from availability. For example, a system may still be counted as available if it is in waiting mode and has a bounded waiting time. Pfleeger *et al.* constructed an overall description of availability consisting of five goals [64]. Following this description, a system is said to be available, if:

- There is a timely response to our request.
- Resources are allocated fairly so that some requester are not favored over others.
- The system involved follows a philosophy of fault tolerance, whereby hardware or software faults lead to graceful cessation of service or to work-arounds rather than to crashes and abrupt loss of information.
- The service or system can be used easily and in the way it was intended to be used.
- There is a control mechanism that ensures concurrency. That is, simultaneous access, deadlock management and exclusive access are supported as required.

2.1.2 Definitons in IT Security

How important a working security process is can easily be seen by looking at various security reporting news sites on the web. Almost every day, reports of hacks, password leaks and other malicious events are published. These attacks affect the privacy of millions of people as the following examples indicate.

During a relative short period of time, from June 2012 to March 2013, several large online community sites have been compromised. In each case, password hashes or personal information about the users have been stolen: LinkedIn, a social network for people in professional occupations, 6 million password hashes stolen [1]; LastFM, an online music community, 2.5 million password hashes stolen [2]; eHarmony, an online dating site, 1.5 million unsalted password hashes stolen [3]; Gamigo, a news portal and community for online gaming, 11 million password hashes stolen [4]; Twitter, a microblogging site, $250,000$ password hashes and user information stolen [5]; Evernote, an online service for note taking, unknown or unpublished number of password hashes and user information stolen [6].

For a better overview, these impressive numbers have been aggregated into table 2.1.

Other forms of security vulnerabilities are common too, for example a serious vulnerability in HP printing devices [7] was detected. It could allow remote attackers to gain access to sensitive

[1] http://blog.linkedin.com/2012/06/06/updating-your-password-on-linkedin-and-other-account-security-best-practices/ (last accessed: 25.09.2014)

[2] http://www.last.fm/passwordsecurity (last accessed: 25.09.2014)

[3] http://www.h-online.com/security/news/item/Millions-of-Last-fm-passwords-leaked-1613641.html (last accessed: 25.09.2014)

[4] http://www.h-online.com/security/news/item/11-million-passwords-leaked-from-online-gaming-platform-1651198.html (last accessed: 25.09.2014)

[5] http://blog.twitter.com/2013/02/keeping-our-users-secure.html (last accessed: 25.09.2014)

[6] http://blog.evernote.com/blog/2013/03/02/security-notice-service-wide-password-reset/ (last accessed: 25.09.2014)

[7] http://www.crn.in/news/security/2013/03/12/hp-printer-flaw-enables-remote-attacks-data-access (last accessed: 25.09.2014)

Community	# of stolen data
Gamigo	11,000,000
LinkedIn	6,000,000
LastFM	2,500,000
eHarmony	1,500,000
Twitter	250,000
Evernote	*unpublished*

Table 2.1: Amount of stolen user information from online community sites.

data. Even governmental institutions are not safe from hacks: The Reserve Bank of Australia has admitted a security breach of their systems [8]. The attackers tried to gain access to internal information, but did not succeed, according to the bank.

In order to deal with all the activities threatening the security of a system, we need clear definitions, starting with the terms *vulnerability* and *threat*. Bishop [10] gives a definition of a *vulnerability* being:

> A weakness that makes it possible for a *threat* to occur.

Whereas a *threat* is defined as:

> A potential occurrence that can have an undesirable effect on the system assets or resources. It is a danger that can lead to undesirable consequences.

Generally speaking, a security threat is a breach of confidentiality, disruption of integrity or denial of service. Threats may have various origins: from either outside or inside some system boundary, from authorized users or unauthorized users, or many other sources are possible too [11]

In his book, Bishop refers to an internet draft called ”Security Architecture for Internet Protocols: A Guide for Protocol Designs and Standards” by Rober W. Shirey in November 1994 [76]. This draft never made it into an Request For Comments (RFC) document and was therefore removed after a fixed period. In this draft, Shirey proposed a partition of threats into four broad classes: *disclosure*, *deception*, *disruption* and *usurpation*.

Disclosure Sometimes called ”interception”, this class contains actions that try to gain unauthorized access to assets. An example is *snooping*, which is an unauthorized and passive interception of information. It suggest that some entity is listening to communications, browsing through files or other system information. A very simple form of snooping is listening to a secret conversation by eavesdropping through the door. Another form of snooping is *wiretapping* in which a network is monitored and all information going through this network can be intercepted. [11]

Deception Deception in general is the acceptance of false data. The preceding step to a deception threat is a *modification* where an unauthorized entity not only accesses but tampers with an asset. An example of a deception scenario is where some party relies on data to determine which action to take but the data has been modified by an attacker. If the incorrect information is accepted as correct, then the deception was successful and is a potential threat to this party. [11]

[8] http://www.rba.gov.au/media-releases/2013/mr-13-05.html (last accessed: 25.09.2014)

Another example is *masquerading* or *spoofing*, the impersonating of one entity by another. The threat occurs by convincing the victim of the masquerade into believing that the entity with which it is communicating is a different entity. This can be a user that tries to log into a remote system but instead reaches some other system that claims to be the desired one. [11]

While the example of wiretapping from above is typically a passive form of snooping, there exist also actions that can be classified as active wiretapping. A prominent example is the *Man-In-The-Middle (MITM)* attack where an attacker intercepts messages from the sender. He can then read the message and even modify them before sending them onwards to the actual receiver.

Disruption If an asset becomes lost, unavailable or unusable to its users this is a form of disruption or *interruption*. Several nuances of the interruption of an asset exists, some of then will be mentioned here. *Denial of receipt* is an attack with a false denial about an entity receiving some information or message. This can be the case when a customer complains to the vendor to have not received some ordered product yet and demands a new shipment. Suppose the customer did actually receive the products but simply claims otherwise, then this is a case of denial of receipt. A similar but still different action of disruption is the *delay*, a temporary blocking of a service. During normal operation the request to a system requires some expected time t to be handled and answered. If an attacker can force the process to take more time than t he has successfully delayed the execution. [11]

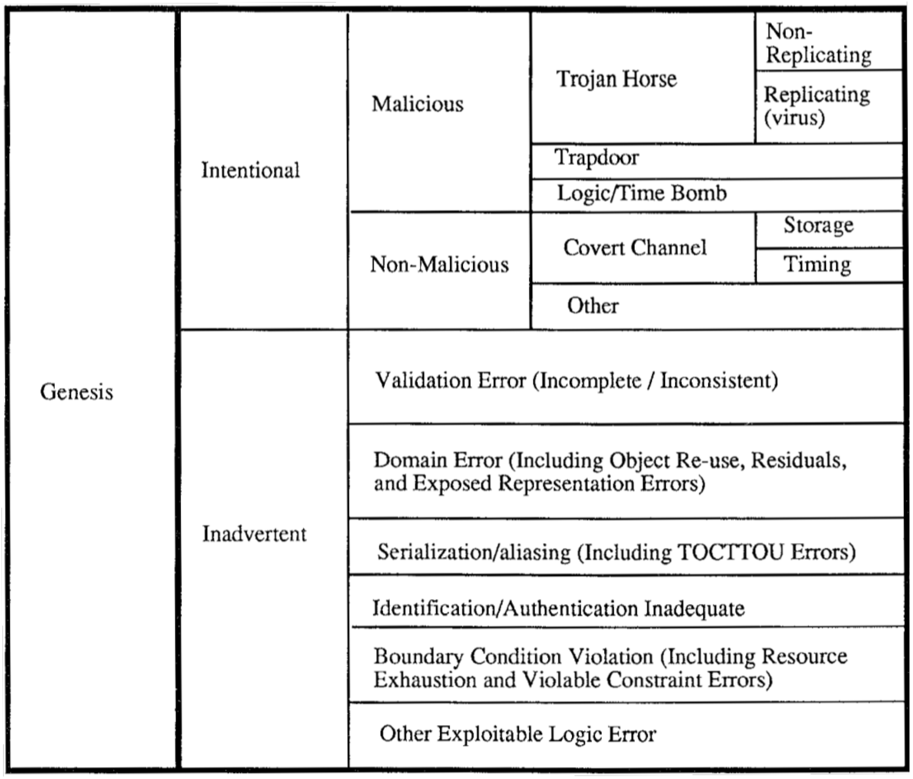
The most famous form of disruption is the *denial of service* which is a long-term interruption of an service or asset. An attacker has several points of actions to chose from. The denial can either occur directly at the source by preventing the server from obtaining resources required for the processing of the requests (e.g., by erasing an important file). It can also occur at the destination by preventing the client from receiving messages from the server (e.g., by misconfiguring the firewall rules). Finally the denial can be along the intermediate path between client and server by discarding message from one or both of the communication members (e.g., by unplugging one from the network). [11]

Usurpation Usurpation is present when an unauthorized party has taken control of some part of a system. This action often comes along with some action from another class. A classical example is the Man-In-The-Middle attack where the attacker can not only read all messages, he can even modify them and thus take full control of the communication.

Masquerading is also often the first step to an usurpation attack. Suppose a user logs into a system which he believes is the one he desires. However, the system itself or the way to the system has actually been modified by a third party. This leads to the situation where the user, in good believe, is connected to a system controlled by someone other than he thinks. All the actions the user takes and all the commands that are executed can then be modified by the attackers to their wishes. [11]

2.1.3 Types of Attacks

This section will explain some practical attacks that are mentioned in this thesis at some point. As mentioned in the previous sections, an attack always targets a weakness in the system. To obtain an overview of the security flaws, Landwehr *et al.* introduced the ”Taxonomy of Computer Security Flaws” [45]. In their work they introduce three taxonomies, one of them is based on the genesis of the security flaws (how did the flaw enter the system). The taxonomy is illustrated in figure 2.2. Of special interest for this thesis are two classes inside the *inadvertent* branch:



Genesis	Intentional	Malicious	Trojan Horse	Non-Replicating
				Replicating (virus)
			Trapdoor	
			Logic/Time Bomb	
		Non-Malicious	Covert Channel	Storage
				Timing
			Other	
	Inadvertent	Validation Error (Incomplete / Inconsistent)		
		Domain Error (Including Object Re-use, Residuals, and Exposed Representation Errors)		
		Serialization/aliasing (Including TOCTTOU Errors)		
		Identification/Authentication Inadequate		
		Boundary Condition Violation (Including Resource Exhaustion and Violable Constraint Errors)		
		Other Exploitable Logic Error		

Figure 2.2: A taxonomy of security flaws by Landwehr *et al.* . [45]

First of all, a major risk to all applications are *validation errors*. A weak validation of input data can lead to an attack where untrusted data is sent to an interpreter, called an *injection* attack. Attacks that belong to the class of injection are: *SQL Injection*, *Path Traversal* and *Cross Site Scripting*. These type of attacks are the most common security risks according to the ”Open Web Application Security Project (OWASP) Top 10” awareness document from 2013 [67].

The second class that will be mentioned here is the class of *identification/authentication inadequate*. These flaws permit the execution of a protected operation without proper checks of the identity and authority of the invoking user [45]. A type of attack that exploits such a flaw is *Session Hijacking*

In the following sections the mentioned attacks will be presented shortly. Starting with the three attacks belonging to the class of injection attacks and following with the session hijacking attack.

SQL Injection This attack belongs to the class of injection attacks where the attacker tries to modify a request in such a way, that the included code will be executed on the target machine. A Structured Query Language (SQL) injection is a specialised kind of injection. The goal of the SQL injection is to insert arbitrary code into a string that is eventually executed by the database. The injected code is for example a part of a database query coded in the SQL statement. The attacker thereby tries to access information inside the database to which he is not authorized to, making this action a *disclosure* attack. The vulnerability that enables this threat is due to a very common programming mistake. The program (often a website) passes user input to a database without doing any kind of input checking. A simple example is :

```
$name = "foo; DELETE FROM users;";
mysql_query("SELECT * FROM users WHERE name={$name}");
```

Listing 2.1: A small example presenting an SQL Injection.

The function `mysql_query` contains the database query and may ask the user for his name. The malicious user, however modifies the query in such a way, that after the selection of the user ”foo” a second query is executed, deleting all user data. [34]

Path Traversal The path traversal attack targets services that offer files for downloading, like web servers do. Every web server has a logical root directory corresponding to some fixed directory on the target machine, most commonly `/var/www/`. Every directory contains a special subdirectory with the name ”`..`” which is a pointer to the parent directory. Using this knowledge and a badly coded web server, a crafty attacker can request any directory on the file system of the server. By multiply adding ”`..`” to the file request he can get down to the root directory of the file system. This hypothetical web server would send an attacker his system user password file when asked for the following URL:

```
http://example.com/../../etc/shadow
```

Listing 2.2: An example URL that can cause a path traversal attack.

Again, this vulnerability is enabled through a program that does not validate the user input and accepts every command as is [20]. With a path traversal an attacker tries to access data to which he has no authorization, classifying this action as *disclosure* attack.

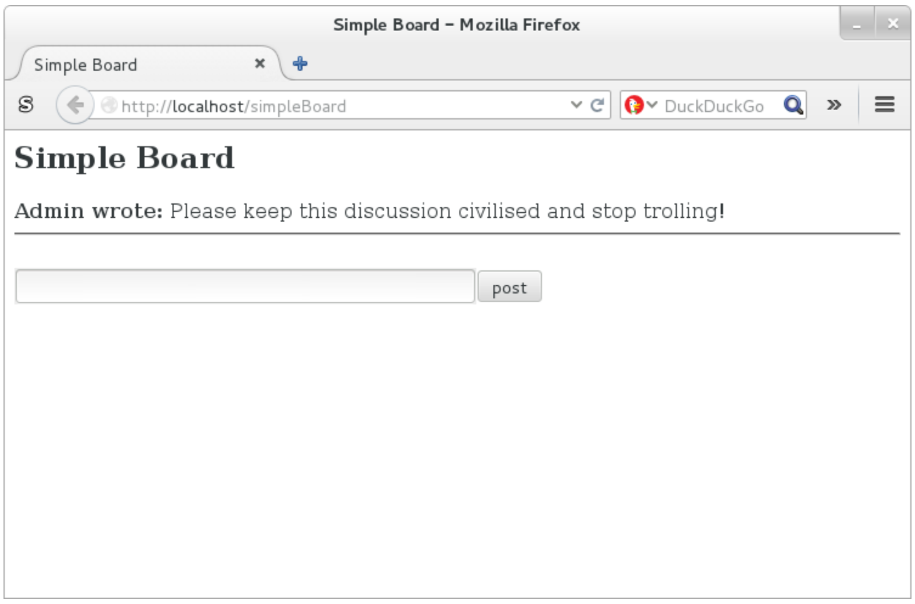
Cross Site Scripting Another injection attack is Cross Site Scripting (XSS). In distinction to the injection attacks mentioned previously, the target of XSS is not the web server directly but focuses on attacking the client. The goal is to inject script code into a web application that is later visited by a user. The modified code is transfered to the client and executed inside the browser. The damage cause by a successful XSS can vary widely, starting with annoying, but harmless pop-ups. However, a skilled attacker can use this technique to hijack session, gain access to restricted content stored by a website, execute commands on the target and even record keystrokes. [17]

A typical script code commonly used in testing for XSS vulnerabilities is the javascript code:

```
<script> alert("You have been hacked!") </script>
```

Listing 2.3: Typical code used for testing XSS vulnerabilities.

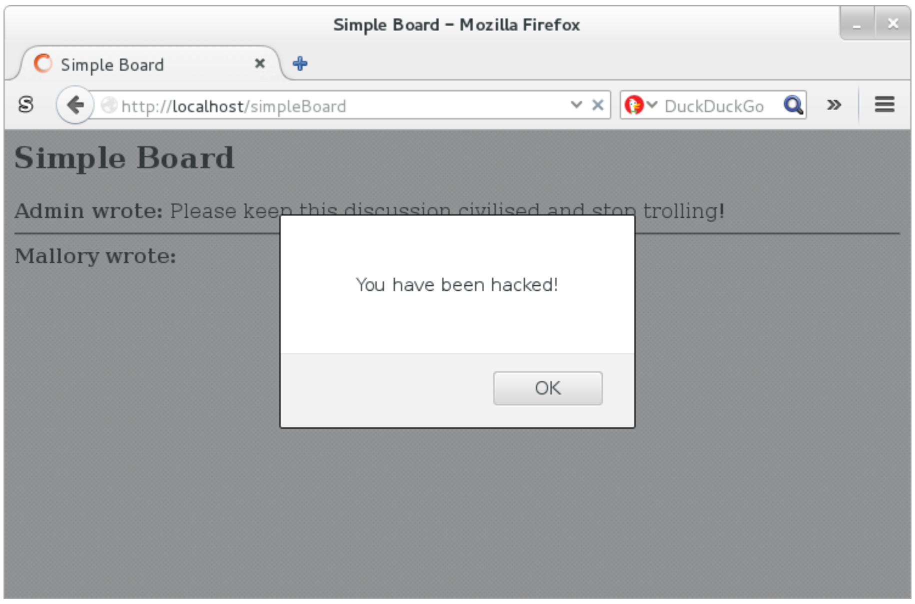
If this little code snipped is placed inside a web application, then every visitor of this page receives a small pop-up with the message ”You have been hacked!”. This situation is illustrated in the two figures 2.3a and 2.3b. Figure 2.3a shows a simple discussion board, implemented as web application. Users can add their comments via the input text field, the inserted text is stored in the backend and displayed to all visiting users. However, the user ”Mallory” found a vulnerability in the application and inserted the code from listing 2.3 into the input field. When sent to the server, this code will not be validated and will be stored in the backend unmodified. Every user accessing the board will then automatically download the webpage including the inserted code from Mallory. The browsers will trust the code, since it is coming directly from the server and execute it, leading to the pop-up shown in figure 2.3b. By exploiting this vulnerability, Mallory was able to execute code in the browsers of all other users.



Simple Board

Admin wrote: Please keep this discussion civilised and stop trolling!

(a) A simple discussion board application.



Simple Board

Admin wrote: Please keep this discussion civilised and stop trolling!

Mallory wrote:

You have been hacked!

(b) The user Mallory inserted javascript code in the discussion board.

Figure 2.3: An example application with XSS vulnerability.

XSS vulnerabilities are another widely spread flaw in web applications. The ”OWASP Top 10 2013” even dedicates those attack an own category, ranked at position three [67]. Another web project that clarifies the risk of XSS attacks is the ”xssed” project [9]. The purpose of this project is to archive as many sites as possible, that are or have been vulnerable to XSS attacks. To each entry, the full exploit can be accessed, as well as a snapshot of the website containing the injected code. An excerpt of vulnerable sites that were published during the last few month is given in table 2.2.

[9] http://www.xssed.com/ (last accessed: 27.09.2014)

site	published
www.bankaustria.at	29.04.2014
wdt.weather.fox.com	29.04.2014
locate.apple.com	29.04.2014
www.paypal.com	13.09.2013
www.dolby.com	20.06.2013

Table 2.2: An excerpt of vulnerable websites taken from the ”xssed” project website.

Session Hijacking When a user has established a connection to a service and authenticated himself by logging in, some kind of token is generated for this connection. This token is necessary for both sides to identify subsequent packets as belonging to this connection. If someone observes all the traffic of the network used for this communication, he has all the information required to imitate the victim. At some point the attacker may decide to hijack the connection, sending proper packets with a faked source address of the victim to the server. The server will regard this packets as originating from the victim and execute all commands accordingly. [78]

With this method even strong authentication mechanisms can be avoided, since the attacker uses the already authenticated connection. An encrypted communication may prevent or at least complicate this kind of attack. In this case, the attacker additionally has to recreate the encryption mechanisms [78]. This attack uses a form of *disclosure* action to gain all the information about the connection as well as a *usurpation* attack when taking control over the session and executing commands with the authority of the user.

2.1.4 Measures against Threats

To prevent all these threats, vulnerabilities and attacks from happening we can use *control* as a protective mechanism. With control (e.g., an action, a device or a mechanism) it is possible to remove or reduce a vulnerability. In his book, Pfleeger draws a picture (figure 2.4) about threats, vulnerabilities and control. Illustrated is a pool of water, that is held in place by a wall keeping the space behind it dry (the *asset*). However, this wall is punctured by a hole, referencing a *vulnerability* in the system. If the water rises further, this vulnerability will lead to a *threat* of water leakage, threatening the dry space. The man placing his finger in the hole, ensuring there is no water flow, is the reference to a *control* of the *vulnerability*.

In general the relationship between threat, control and vulnerabilities is defined as [64]:

A *threat* is blocked by *control* of a *vulnerability*.

The control over the vulnerabilities is performed by one or many security mechanisms. Bishop categorises them into three classes based on their strategy [11]: *Prevention*, *Detection* and *Recovery*. Mechanisms belonging to these classes either prevent an attack, detect an ongoing or past attack or execute recovery methods after or during an attack. All three strategies may be used in combination or separately.

Prevention *Prevention* of an attack is the favoured strategy. An attack is prevented if it is ensured that the attack will fail, so that no harm to the protected systems is done. A crude example of preventing an attack on a system over the network is disconnecting the system from the network. This may be a valid solution in some cases, but will typically disable the systems intended operation. Preventive mechanisms are often very cumbersome and interfere with the usage of the

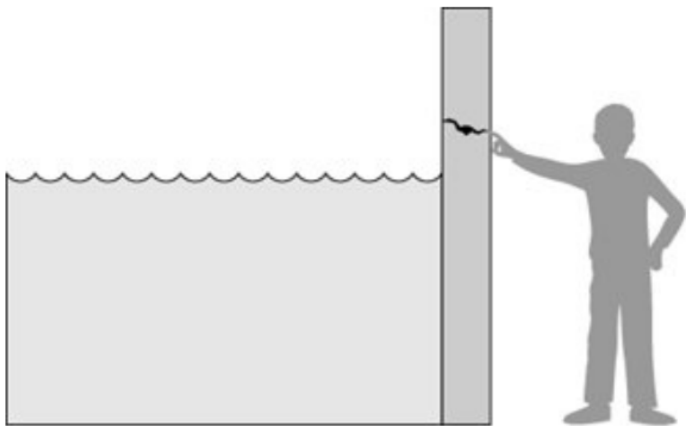


Figure 2.4: A picture about threats, vulnerabilities and control. [64]

system. However, there are some solutions that have become widely accepted. The following section will give some examples of widely used vulnerability control system using the strategy of prevention. [11]

A very important part of securing a system is *access control* [5].

Detection If an attack cannot be prevented, another control strategy is *detection*. Detection mechanism accept the fact that an attack will occur. The goal is to detect an ongoing or past attack and report it. The attack may be monitored, to provide data for further analysis. An example of a simple Intrusion Detection System (IDS) is one that gives a warning if a user enters an incorrect password more than three times. The system will not hinder the user from entering his password, but the warnings will report an unusual high number of mistyped passwords for this user. Another example of an IDS is a system that monitors the network and scans the traffic for outgoing spam messages. This detection mechanism essentially tells the operator that a particular system is compromised. [5]

Recovery The third strategy of security mechanism is *recovery* and has two forms. The first is to stop an ongoing attack and assess and repair any damage that was done. For example, if an attacker deletes a crucial file, a recovery mechanism would be to replace the lost file with a replacement from a backup source. Since the characteristics of each attack can be very different, the recovery is far more complex than this example indicates. However in all these cases the systems functionality will be affected by the attack. [11]

The second form of recovery is a very powerful but quite difficult to implement because of the complexity of computer systems. In this scenario, the attacked system will continue to function correctly even during an ongoing attack. These recovery system are based on techniques of fault tolerance and techniques of security and are typically used in safety-critical environments. The main difference to the first form is, that at no point does the system function incorrectly. [11]

2.2 Software Testing

Software has undergone testing for almost as long as computer programs have been developed. Today several formal definitions of testing exist:

> Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. (Hetzel [30])

This definition addresses an intuitive view of testing: there exist some specifications about how the system should behave, and the testing process should confirm that these requirements are met. This approach is also known as *positive testing*, or *functional testing* [48].

Another definition of testing addresses a different view, known as *negative testing*:

> Testing is the process of executing a program or system with the intent of finding defects. (Myers [59])

This definition does not consider the requirements of the software. It introduces the notion of actively looking for defects outside of the specifications, in order to find errors that may compromise the successful operation or usefulness of the system. In practice, a testing process will combine both positive and negative testing approaches. [85]

Besides the definition of testing, we require some more definitions about security errors in order to characterise them correctly. The Institute of Electrical and Electronics Engineers (IEEE) defines the terms mistake, fault, error and failure as follows [68]:

> **Mistake:** An action done by a human that leads to an incorrect result.
>
> **Fault:** An incorrect part of a computer program. This could be a incorrect computation step, some process of data definition. ”Defect” is often used as a synonym for faults.
>
> **Error:** The difference between the specified value and the value computed by the program.
>
> **Failure:** The inability of a program to execute its required functions to comply with its specifications.

In general, the purpose of testing is a destructive process of trying to find the errors in a program. A test case is regarded successful if it causes the program to fail and thereby furthers the progress of the development. The best way to oppose the claim of an error free program is done by refuting this statement by finding some error in the program. [59]

Regarding the properties of errors, some defects are subtle and can be difficult to detect. This fact causes software testing to often be an expensive task. Not testing, however, can have an even larger effect on an organization’s business. The impact of a system not performing correctly can easily result in financial losses for the providing company. Among other reasons, this may happen through the loss of working time to recover the system or through the loss of prestige and public confidence. There even are examples where the failure of a system has resulted in the software provider going completely out of business. [85]

Edsger Dijkstra stated in a famous work [16], that to show the presence of bugs software testing can be a very effective way. However, to show the absence of bugs it is hopelessly inadequate. In fact, even with mathematical methods (e.g., formal verification, see below) it is impossible to ensure that but the simplest programs are provably free of defects. Another problem is, that resources for

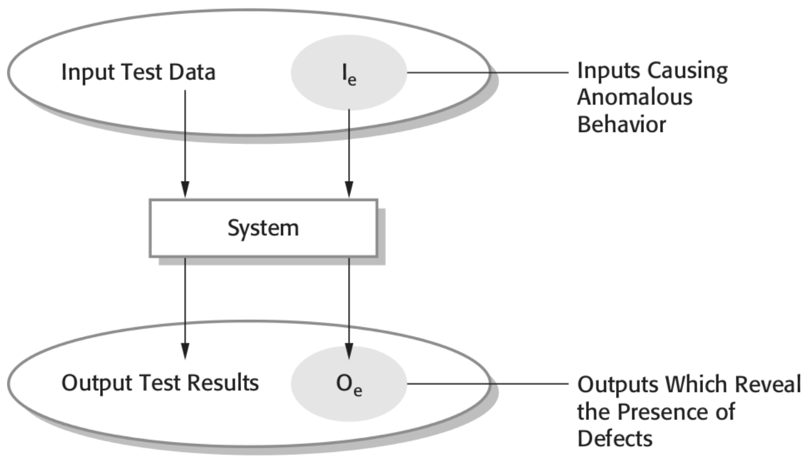

Input Test Data
I_e
Inputs Causing Anomalous Behavior
System
Output Test Results
O_e
Outputs Which Reveal the Presence of Defects

Figure 2.5: An illustration of a black box test. [79]

testing are finite and typically in short supply. With both problems combined, testing appears to be very inefficient process and adequate testing seems unlikely. Software testers have the difficult task to make the testing process – against all odds – as efficient and effective as possible. [85]

Some important testing strategies, that rely on the available knowledge about the System Under Test (SUT), are the complementary techniques *black box* and *white box* testing. If the test analyst has information about the internal structure of the software, the designed tests based on that knowledge are called white box tests. On the other hand, if no internal information is available, the tests are called black box tests. The design of these tests is therefore solely based on the external behaviour of the system [85]. This thesis is primarily focused on black box testing.

2.2.1 Black Box Testing

The name black box testing derives from the notion of viewing the program as a black box. Another name for black box testing is *data-driven* testing. The goal is to be complete uncertain of the internal structure and behaviour of the program. The only thing known to the tester are the specifications from which the test data is derived. An example that illustrates black box testing is given in figure 2.5. The system can be seen as a black box which does not reveal its contents. The test data is given the system as input and it generates some output. The goal of this testing approach is to send specially crafted input data I_e that causes anomalous behaviour in the system, which can be detected through the output O_e. [79]

In order to find all errors in the program, it is important to have an *exhaustive input testing* process. By using every input for testing you will eventually end up finding all errors. However, trying this will quickly end in a scenario with an impossible amount of test cases. It is not only important to test all valid input, but all *possible* input values, which will probably be an infinite amount of test cases. This leads to two implications about black box testing: first you literally can not test a program to guarantee that it is error free. The second implication is that you have to design a black box test to be economically efficient. The objective is to maximize the number of errors found but limiting the amount of test cases to a finite, manageable number.

Some relaxation to the problem would deliver a black box test approach with some additional information about the program. It is not required to have full insight to the program code, just enough to make some valid assumptions about the program. [59]

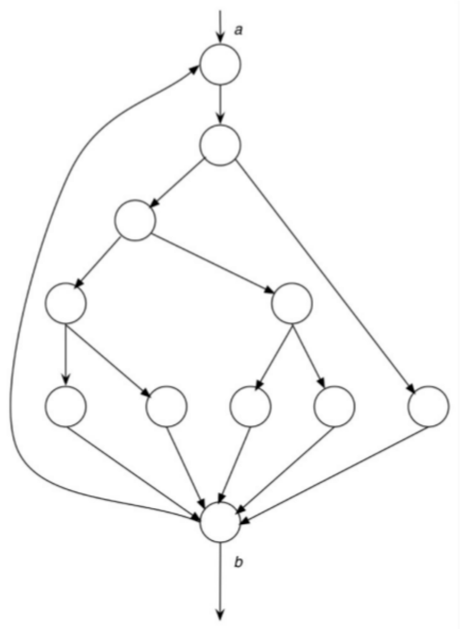


Figure 2.6: The control flow graph of an example program. [59]

2.2.2 White Box Testing

The second testing strategy in this list is white box testing, or sometimes called *logic-driven* testing. It permits the tester to examine the internal structure of the program. The goal is analogously to the goal of black box testing: trying to get an exhaustive input testing process, causing every statement in the program to be executed at least once. Similar to the attempt in the black box testing approach, it is not difficult to show that this method is highly inadequate. The method to do this testing approach is called *exhaustive path testing*. It runs test cases in order to execute all possible paths of control flow inside the program. [59]

The problem is, that even a very simple program can have an incredible high number of different logic paths. Myers delivers a very simple example program (see figure 2.6 for the control flow graph) and calculates the number of paths to be approximately 10^{14} [59]. Even if it would be possible to test every logic path inside a program, it is in no way guaranteed to be free of errors. For example, path testing cannot test a program against its specifications. If the path test returned with no detected error, the program could still be simply the wrong program.

A very interesting technique that belongs to the class of white box testing is *formal verification*. In fact this is not a testing but a prove technique. A formal approach is executed by assuming *precondition* states and checking the *postcondition* states after the program run:

$$\{Preconditions\}$$
$$|$$
$$\text{Program}$$
$$|$$
$$\{Postconditions\}$$

To analyse the program, the tester makes an assumption that there exist flaws in the system. The tester determines the state in which the vulnerability will arise, which is the *precondition* of the program. Afterwards, the tester puts the system into the defined state and analyses the program execution. The analysis will give information about the resulting state of the system, which is called the *postcondition*. This postcondition can be compared to the security policy of the system. If there is an inconsistency, the hypothesis of an existing vulnerability is correct. [10]

Concluding can be said that neither of the two testing strategies proves to be useful on its own, because both are infeasible. It would be necessary to combine elements of black box and white box testing to derive a reasonable testing strategy. [59]

2.2.3 Gray Box Testing

As the previous sentences already suggested it is possible to further loosen the tight definitions of black and white box testing. The approach that combines techniques of white box and black box testing is called *gray box* testing. Black box testing typically looks at the expected behaviour of an application from the point of view of the user. White box testing has all knowledge of the internal data of the program and therefore looks at testing from the point of view of the developers.

Gray box testing combines elements of both black box and white box testing. It evaluates application design in the context of the interoperability of the systems components. This strategy consists of methods derived from the knowledge of the application internals and the environment with which it interacts. With this possibilities gray box testing can reveal problems that are not easily discovered by either of its parents: Problems such as end-to-end information flow and distributed hardware/software system configuration.

However, gray box testing alone is not the sole solution to all testing problems. The key is to have a mix of testers understanding every aspect of the program, of the internal processes and the interactions with its environment. [60]

2.3 Testing of IT Security

As stated in section 2.2, security related defects can not be detected by functional testing. Thus, security tests need to be planned and executed separately. The testing process can take place alongside functional testing, but it has almost an opposite focus. [48]

The target of security testing is similar to functional testing, though: finding errors in the system that lead to security vulnerabilities. Security vulnerabilities manifest themselves as additional behaviour of software, something extra the software does that was not originally intended [83]. The target of security testing is therefore to find every unspecified functionality of the tested software. This additional functionality could be a vulnerability that may allow a threat to occur. Several techniques have been developed in the attempt to reach this goal.

2.3.1 Fuzz Testing

One technique used for testing the effectiveness of security measures is *fuzz testing*, for simplicity reasons often called *fuzzing*. The purpose of this technique is to send anomalous data to a system in order to find security vulnerabilities. Fuzz testing is defined as a highly automated, negative testing approach. [48]

Similar to the software testing strategies black, gray and white box testing, fuzz testing tools (also known as ”fuzzers”) can be classified based on the knowledge given about the SUT. A black box fuzzer has no further information about the internal structure of the tested system. Figure 2.5 illustrated a general black box approach for software testing and is also applicable for black box

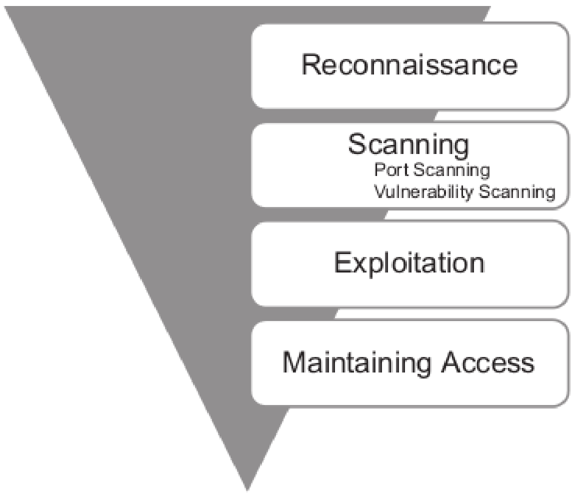


Figure 2.7: All phases of a complete penetration test. [17]

fuzz testing. The fuzzer sends input data to the system and scans the output in order to find defects in the tested software. [81]

White box fuzzing can infer further information about the SUT by analyzing its source code. This additional information can, for example, be used to automatically generate test data. Godefroid [23] presented an approach where a white box fuzzer is enhanced with a grammar-based specification of valid inputs. Based on this context free grammar, a constraint solver can then automatically deduce test cases for the fuzzer. [23]

As before, a combination of both approaches is gray box fuzzing. It uses the testing approach of black box fuzzers with some additional run-time information about the SUT to improve testing. [81]

2.3.2 Penetration Testing

Penetration testing is a another testing technique and allows a view on the system through the eyes of an attacker. It can be defined as a legal and authorized attempt to locate and successfully exploit a system for the purpose of making this system more secure. The difference to many other techniques is, that penetration tests not only search for, but actually exploit the vulnerabilities in order to prove that a security issue exists. A penetration test is even more useful if the testers give specific recommendations for addressing and fixing the discovered issues afterwards. [11].

Depending on the contract, the penetration test can take on one of two forms. The first type is an attempt to violate some specified controls in the security system. An example goal for such a test would be to gain read and write access to a defined file on a specific server. The second type of penetration test does not have such a well defined target. Instead the goal is to find as many vulnerabilities in the system as possible within a given period of time. [11]

In general, a penetration test is executed from an attacker's point of view. By simulating an attack, the security mechanisms can be tested in the environment in which an assumed attacker would function. Since different attackers come from different environments, this suggests a layering of penetration tests comparable to the black, gray, and white box tests presented in section 2.2 above. A possibe layering is: [11]

- *External attacker with no knowledge of the system.* At the lowest level, the testers have as little information as possible. They have knowledge that the system exists and can identify it, once found. Gathering more information is thus an additional task for the testers, which they can handle by e.g., social engineering or network scans.
- *External attacker with access to the system.* At this level, the testers already have access to the system. However their privileges are limit to those available to all hosts on the network. From this initial position the testers have to launch their attack. For example, the testers can try to get access to an account from which they can achieve their goals.
- *Internal attacker with access to the system.* This level can be compared to a white box testing approach. The testers have an authorized account on the system and a good knowledge about its internal structure. Based on this knowledge, the testers can develop and execute attacks.

A complete penetration test includes all the steps that any attacker would take. These steps typically are: *Reconnaissance*, *Scanning*, *Exploitation* and *Maintaining Access*. Figure 2.7 visualises the four phases of a penetration test. The inverted triangle represents the methodology going from very generic to very specific. At reconnaissance every piece of information is stored about the target where at the maintaining access phase very specific actions are taken. [17]

Reconnaissance is the phase of information gathering. The attacker investigates their target using publicly available information. Every detail and every piece of information is collected and stored and possible targets are located. In comparison to the other phases, this is the least technical one. Some examples for reconnaissance techniques are: web searches, ”whois” database analysis and Domain Name System (DNS) querying. [78]

In the second step, scanning, the attacker has collected some vital information about the infrastructure of the target, like DNS names and Internet Protocol (IP) addresses. Armed with this knowledge, the attacker may then scan target systems looking for openings. A careful mapping of the network infrastructure can help to determine the critical hosts, routers and firewalls. Following this thorough network scan, the attacker wants to discover potential entry points into the host by scanning the machine for open network ports. By then the attacker will have a good understanding of the services of the system and can try to discover vulnerabilities in these programs, by using automated vulnerability scanners, for example. [78]

Exploitation is the phase where the attacker tries to gain control over the previously identified, vulnerable targets. The particular approach can have many different forms, starting from using readily available exploit tools to highly sophisticated methods. The course of actions taken depends heavily on the situation, requiring for example local attacks directly on installed application or the operating system. Attacks can also be executed on a network, sniffing for information or manipulating communications. [78]

After gaining unauthorized access to a system, the attacker will try to maintain the access. The ways opened in the third phase are often only temporary, thus the final phase consists of converting the temporary access into a permanent one. To achieve this goal the attacker has an arsenal of techniques based on malicious software at disposal. Examples of such software are trojan horses, backdoors, bots and rootkits. [78]

This description of a complete penetration test includes all the steps any attacker would take in an attempt to control the target. Which of these steps the penetration test will include depends on the mutual agreement of the employer and the testers. For example the fourth phase, maintaining access, can seriously compromise an existing system and may be omitted in a penetration test.

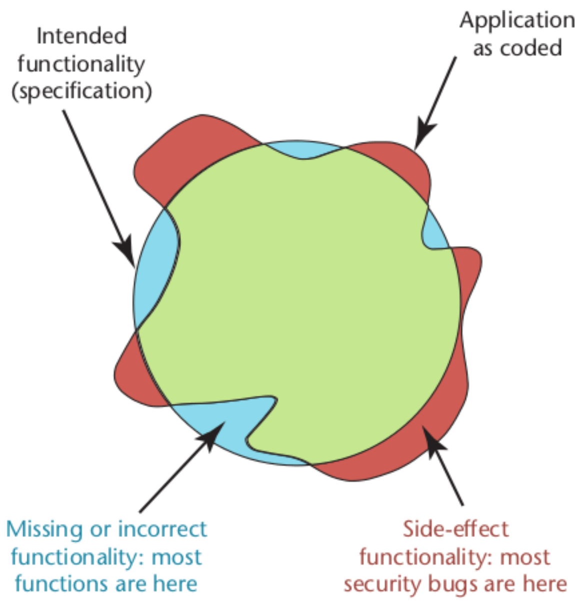

Intended functionality (specification)
Application as coded
Missing or incorrect functionality: most functions are here
Side-effect functionality: most security bugs are here

Figure 2.8: Intended and implemented functionality in software. [82]

2.4 Challenges of Security Testing

Software testing, as described in section 2.2, has developed into a very good and useful discipline at verifying functional requirements. Several types of bugs can not easily be detected through functional testing, though. Even a software that was frequently tested against its requirements and is considered to behave perfectly correct, according to its requirements, may not be considered secure. This is due to the fact that the application may perform some additional tasks that were not specified or even intended. Flaws like this will not be detected by functional testing, because the test cases are designed to look for the presence of some correct behaviour and not the absence of additional behaviour. [82]

As an example, a typical test case may look like: ”apply input A, and check for output B”. Suppose that during the calculations for output B, the application produces some other output C. If the output is some obvious action, like a dialog box popping up, the tester will certainly notice. However, the action can even be something more subtle like the writing of a file or the opening of a network port. Such a behaviour will probably never be detected by the testers, although it will occur during every test case and may result in a dangerous side-effect. [82]

Thompson describes the situation of intended and implemented software functionality, which is illustrated in figure 2.8. The circle represents the intended functionality of a software as given by its specification. In a perfect world the shape of the implemented software would fit right into the circle, but in practice this is hardly the case. The amorphous shape represents the actually implemented functionality of the application. The areas in the circle that are not covered by the implementation are behaviour of the software that was implemented incorrectly, thus a typical software bug. These flaws in the application can be detected by functional testing. On the other hand, the areas of the amorphous shape that lie outside the circle represent behaviour of the implemented software that was not intended or specified. This behaviour is potentially dangerous software functionality. [82]

3 Artificial Intelligence in IT Security

”How do we think?” – This question intrigues us humans now for several thousands of years. Still we try to answer the questions about how it is possible that our brain, which is just a handful of matter, can perceive, understand, predict, and manipulate a world far more complicated than itself. Several research fields try to find answers to that question. One of them is Artificial Intelligence, which not only tries to give answers to that question, but also to go even further. It tries to *build* intelligent entities. The work on AI is divided into several subfields, for example learning and perception. [72] Some applications of AI are: robotics, speech recognition, planning and scheduling, translation, and many more. [72]

The work of this thesis will concentrate on methods for *Classification* and *Clustering* which are applications of the subfield of *Machine Learning*. These principles can be used in a wide range of IT Security applications. For example automated classification, if a suspicious network packet is regular traffic or if it is coming from a individual with ill intentions.

This chapter will start with a general introduction to the topic of Machine Learning. Subsequently it will discuss several methods of AI that can be used in combination with IT Security systems. It will start with methods able to classify the input data given: *Artificial Neural Networks*, *Self-Organizing Maps*, *Support Vector Machines* and *Clustering*. Every method will be described and its current usage in the topic of IT Security will be discussed.

3.1 Machine Learning

In general one can say that there are a lot of problems for which no solution exists. The reason is that we either do not yet know how to solve them, or we provably know that it is not possible to write an exact algorithm that solves it. In short, an algorithm is a sequence of instruction that should be carried out to transform a given input to some output. An example for a yet unsolvable problem is face recognition. A solution to this problem has the task to transform the image of a face to a name or any other identification. We do this task effortlessly every day, but since we can not explain *how* we do it we can not write an algorithm for this task. Thus, no algorithm exists so far that can automatically and provable correctly recognise a person from a face image. [3]

However, in many cases it is not necessary to have an algorithm that solves a problem perfectly. In the field of Machine Learning (ML) exist methods that can approximate solutions to a given problems. These methods may not be able to find an optimal solution to the problem, but we will still be able to extract an adequate solution. These ML methods are based on the idea, that what we lack in knowledge, we make up for in data. A definition of Machine Learning by Alpaydin [3] is:

> Machine learning is programming computers to optimize a performance criterion using example data or past experience. (Alpaydin [3])

With the advances in computer technology, we are now able to store huge amounts of data and even access it from physically distant locations. We believe that there is a process that explains the data we observe, e.g., why the image of a face looks like a face. We may not be able to identify

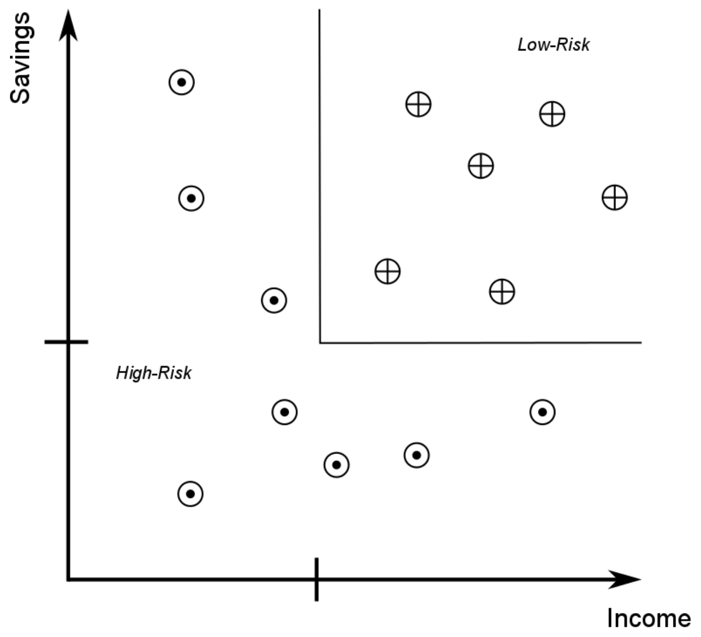


Figure 3.1: The training data for the classification example. (based on [3])

the process exactly, but we believe we can construct an approximation that yields us a good and useful solution.

For example, a face image is not just a random collection of pixels, such an image has a structure. It is symmetric, has two eyes, a nose and a mouth positioned on certain places. By training on sample faces, a learning program can analyse the patterns specific for one person. Later, it can recognise that person by analysing a similar image and drawing conclusions from its previous experience, or technically speaking: from its knowledge base. [3]

Two applications of ML are called *classification* and *clustering*, which will be discussed further. Both methods try to find patterns in a given set of data, but they are separated by one main difference. This difference is mainly located in the training phase. *Supverised learning* methods can access some additional knowledge about the goals of their task. *Unsupervised learning* methods on the other hand are given no such information and have to draw conclusions solely based on the structure of the data itself.

3.1.1 Classification

Classification is an application of machine learning, which belongs to the class of *supervised learning* methods. This means that during the learning phase, the program is guided by some kind of supervisor. The classification program improves its performance by learning mappings from input values to output values. The correct values of these mappings are known and provided by this supervisor. In the face recognition example, the training process would be done by giving the classification algorithm some images of faces and additionally telling it the correct answer. After a few iterations, when the program is trained sufficiently, it can draw conclusions from this stored knowledge and recognise a trained face by itself. [3]

An application of classification is the loaning of a credit from a bank to a customer. The bank is interested in the probability of the case that the customer will default and not be able to repay the money. The institute will collect financial information about the customer, e.g., income and savings, and compare the data to records of past loans. From the knowledge, if these past loans

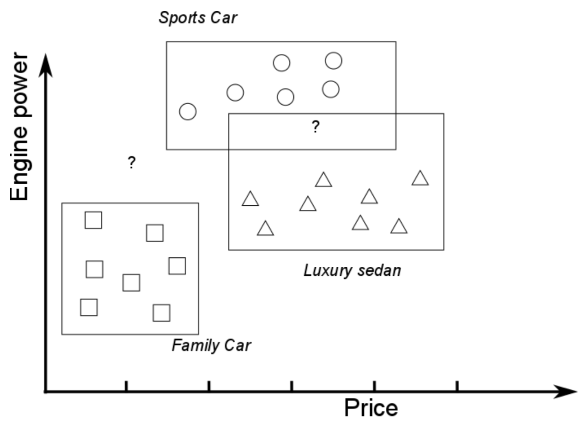


Figure 3.2: A example clustering of input data consisting of three clusters. (based on [3])

have been paid back, the bank can draw conclusion to the present case. The bank will be interested in a *classification* of the customer into, for example two classes: low-risk and high-risk customer. The collected, financial information about the customer will be the input to a classifier program. The task of the program is to assign the input to one of the two classes. Based on the knowledge learned from past loans, represented by the values of α and β, a classification rule may look like:

IF $income > \alpha$ AND $savings > \beta$ THEN `low-risk` ELSE `high-risk`

The rule acts as *discriminant*, separating the training samples into two different classes, as shown in figure 3.1. With a rule like this it becomes possible to make *predictions* of the future. If we assume that the future will not be much different from the past, we can make correct predictions for future instances. For the next loaning decision, the bank can collect the financial information and apply this rule to easily decide whether the customer is low-risk or high-risk. [3]

3.1.2 Clustering

Clustering is a method belonging to the class of *unsupervised learning* methods. This means that there is no supervisor and only the input data is available to the algorithm. The goal of clustering is to find some kind of regularities inside the input data. There probably exists a structure in the data such that some patterns occur more often than other. The clustering algorithm has to detect these patterns.

An example of Clustering is called *customer segmentation*. A company stores demographic information as well as past transactions about its customers. It is interested in the distribution of the profile of its customers, to see what type of customers frequently occur. The clustering method groups customers that are similar in their attributes, providing the company with a natural grouping of its customers. Once such groups are found, the company can develop strategies that are specially designed for specific groups. The clustering even identifies outliers, customers that do not belong to any groups. They may imply a niche in the market that can be further exploited by the company [3]. An illustration of a typical clustering situation is drawn in figure 3.2.

3.2 Artificial Neural Network

The idea of artificial neural networks is based on the realization that a brain of a human, or an animal works in a completely different way than current digital computers do. A brain consists of multiple basic cells, called *neurons* that are connected by structural units, called *synapses*. Each neuron cell is a small calculating unit and has a very slow rate of operation, where an event happens in the range of milliseconds $(10^{-3}s)$, compared to a silicon chip, where events happen the range of nanoseconds $(10^{-9}s)$. However, the brain makes up the lack of speed with its massive number of neurons and connections in between. The human brain, in particular, consists of up to 100 billion neural cells and about 60 trillion synapses, making it a highly complex, non-linear and parallel computer. With this characteristics, it has the capability of performing certain computations (e.g., pattern recognition) many times faster than a conventional, digital computer. [27] [18]

The research on artificial neural networks is based on the knowledge of biological neural networks, trying to model or simulate the functionality. An elementary work on this field did McCulloch and Pitts in their 1943 publication ”A logical calculus of the ideas immanent in nervous activity” [52]. They were the first to design a mathematical model of a neuron as basic element of brains. [18]

3.2.1 Biological Model of a Neuron

As mentioned above, the human brain consists of about 100 billion neural cells. The structure of a biological neuron is illustrated in figure 3.3. Each neuron consists of a cell body and an *axon*. The axon acts a connector to the synapses of the neighbour neurons. Every neural cell in the brain has about 1000 to 10000 connections to other neurons, creating a massive network. [18]

The cell body of a neuron can store some amount of electrical voltage, similar to a capacitor. It is charged by the voltage pulses of its incoming connections from other neurons. If the stored voltage reaches a certain level in the cell, it is again released as a brief voltage pulse, commonly known as *spikes*, over the axons and the synapses to its neighbouring neurons. There, the process is repeated.

An elementary role in the neural network play the synapses, which are positioned on the axons between every two neurons. The most common kind of synapse is a *chemical synapse*. This synapse type is no perfect conductor, since it connects its two ”wires” not directly, but with a transmitter substance. To be able to pass the transmitter substance, the incoming electrical signal has to be converted into a chemical signal at first and then converted back into an electrical signal. The conductibility of the substance depends on various parameters like the concentration or chemical composition of the substance. With this mechanism, a synapse is able to control the interaction between two neurons. [27] [18]

3.2.2 Mathematical Model of a Neuron

A single neuron is the fundamental unit for the construction of a neural network. The design of the mathematical model is directly based on its biological counterpart. To define the mathematical model of a neural network, Haykin identified three basic elements of a neuronal model [27]:

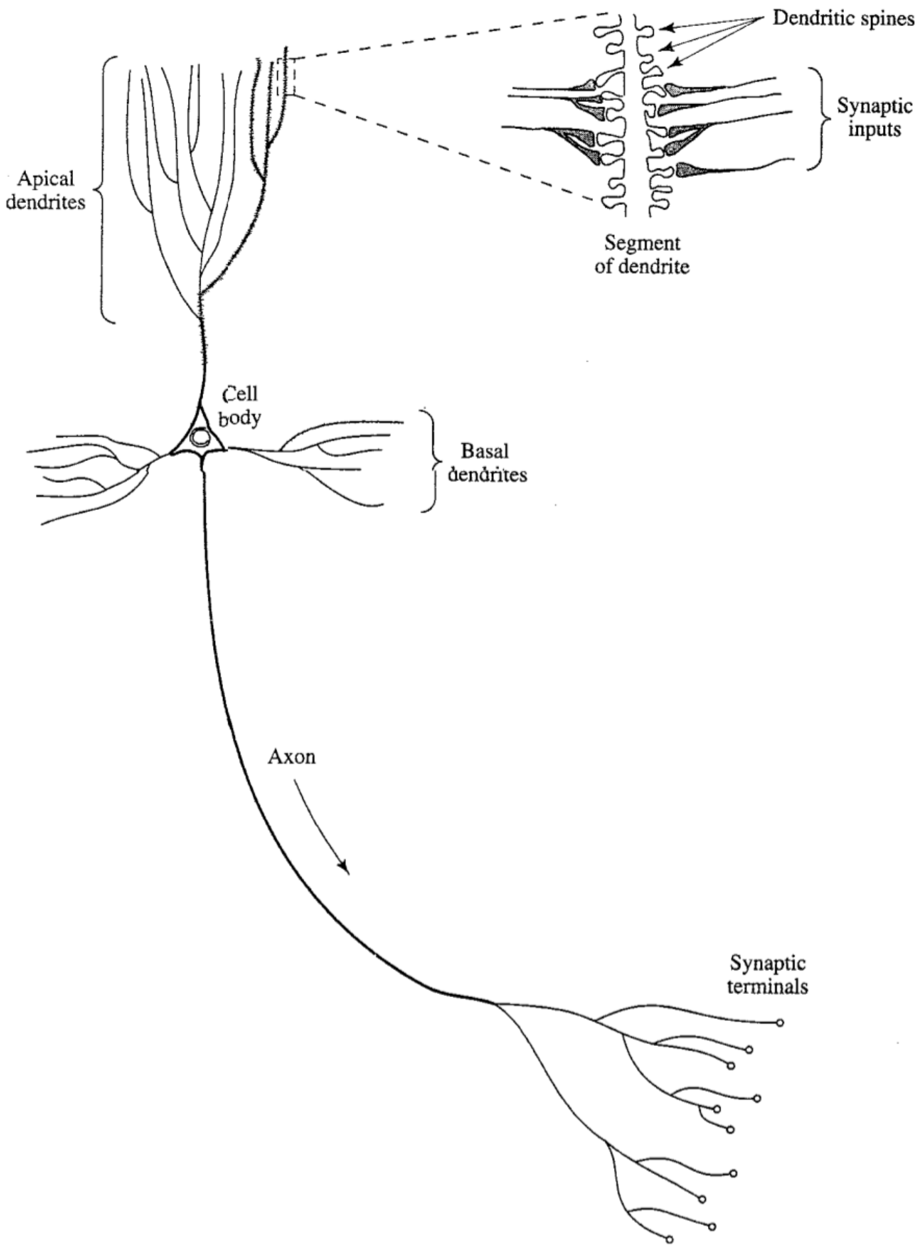


Figure 3.3: The biological model of a neural cell. [27]

1. A set of connecting links with corresponding *weights*. They represent the synapses of a biological neuron, together with their conductibility. For every neuron k, and for every connected synapse j the connecting link has a weight w_{kj}. An incoming signal x_j is multiplied with the weight w_{kj} to the final input of the synapse j.
2. An *adder* function for combining up the input signals of the synapses.
3. An *activation function* that has the task to limit the output of the neuron. Typically the output of the neuron is limited to a closed unit interval of $[0, 1]$, or alternatively $[-1, 1]$.

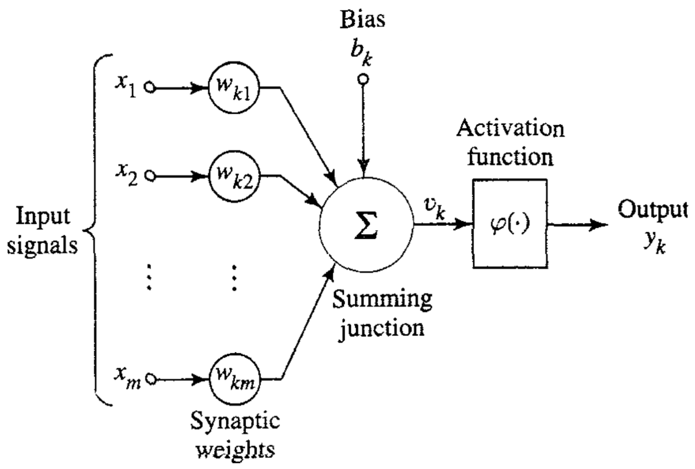


Figure 3.4: A mathematical model of a neuron. [27]

Based on this elements, a model of a neuron is illustrated in figure 3.4. A possible adder function is a simple sum over all the input elements, represented as formula:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{3.1}$$

where $x_1, x_2, \ldots, x_m$ are the input signals and $w_{k1}, w_{k2}, \ldots, w_{km}$ the weights of the synapses of neuron k. An external parameter, the *bias* b_k, can be used optionally to apply an affine transformation to the output u_k. This effect is illustrated in figure 3.5. The formula for the applied bias is:

$$v_k = u_k + b_k \tag{3.2}$$

The activation function is then applied onto the result of the adder function, yielding the final output of neuron k:

$$y_k = \varphi(v_k) \tag{3.3}$$

The simplest activation function is the identity $\varphi(x) = x$, which just passes the values on that are given as parameter. However, this may lead to convergence problems since the identity function is not bounded [18]. The next section describes some activation functions that are bounded.

Activation Function

The activation function represents the function of a biological neural cell to store voltage up to a specific level until it releases the voltage as spike. Three basic types of activation functions can be identified [27] : *Threshold Function*, *Piecewise-Linear Function* and *Sigmoid Function*.

Threshold Function This activation function is also known as *Heaviside Function* and does only return two values, e.g., $(0, 1)$. The output depends on whether the input is larger than a specified *threshold* Θ or not (from [18]):

$$\varphi(v) = \begin{cases} 0 & \text{if } v < \Theta \\ 1 & \text{otherwise} \end{cases} \tag{3.4}$$

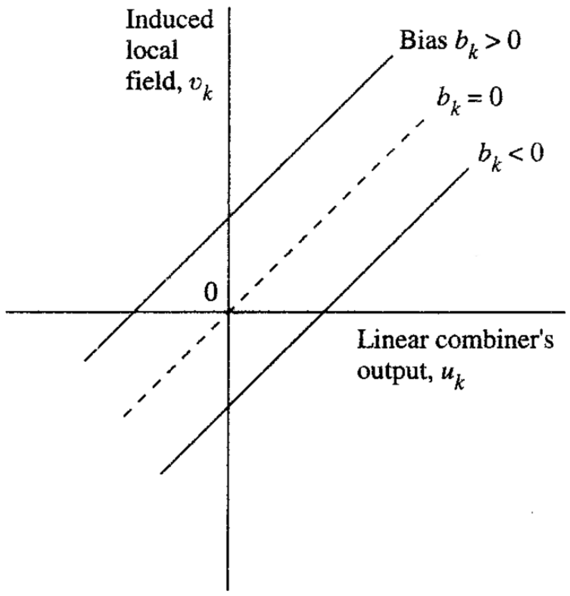


Figure 3.5: The transformation produced by adding a bias b_k. [27]

Applied to the neuron k this gives the function:

$$\varphi(v_k) = \begin{cases} 0 & \text{if } v_k < \Theta \\ 1 & \text{otherwise} \end{cases} \tag{3.5}$$

This function can be useful for a binary neuron, since the activation function of a binary neuron can only return the values 0 or 1. A special case of a binary neuron model is the *McCulloch-Pitts model.* In this model the threshold value Θ is set to 0 and the activation function y_k only returns 1 if the output of the neuron (v_k) is larger or equal than 0.

Piecewise-Linear Function The piecewise-linear function is a combination of a threshold and a linear function [27]:

$$\varphi(v) = \begin{cases} 1 & \text{if} & v \geq +\frac{1}{2} \\ v & \text{if} & +\frac{1}{2} > v > -\frac{1}{2} \\ 0 & \text{if} & v \geq -\frac{1}{2} \end{cases} \tag{3.6}$$

For a defined region, $[-\frac{1}{2}, +\frac{1}{2}]$ in this case, the function acts as linear function. To satisfy the bound properties that are required for activation functions used in neural networks, the function returns a maximum (respectively a minimum) if the input is outside the linear region.

Sigmoid Function The most common form of activation functions used in neural networks is the sigmoid function. It offers a fine balance between linear and non-linear behaviour. An example is the *logistics function* [27]:

$$\varphi(v) = \frac{1}{1 + e^{-av}} \tag{3.7}$$

Where the parameter a defines the shape of the function graph. As a advances to infinity the sigmoid function simply becomes a threshold function. The return value of the function is a continuous range of values from 0 to 1.

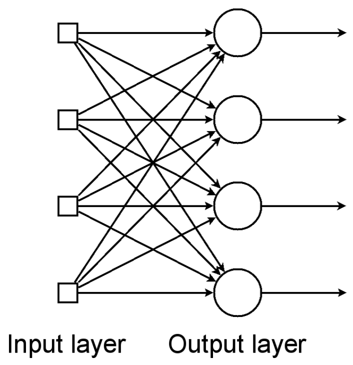


Figure 3.6: An example of a fully connected single-layer neural network. (based on [27])

3.2.3 Neural Network Architecture

So far, the functionality of a single neuron model have been defined. To gain their full potential, multiple neurons have to be connected to a neural network. The neurons are structured into a *directed graph* with at least two different types of neurons. The learning algorithm is a direct consequence of the architecture of the networks, it will be discussed in section 3.2.4. This section explains the different structure types of artificial neural networks.

First of all, the network can be *fully connected* or *partially connected*. In a fully connected network, every node in one layer is directly connected with every node on the adjacent layer. If this is not the case and some possible connections are missing, then the network is called partially connected. [27]. As example, the network in figure 3.6 is a fully connected neural network, since each neuron on a layer is directly connected to each neuron in the next layer. Figure 3.7 on the other hand is said to be partially connected. The neurons in the last layer do not receive input from all neurons in the layer in front.

Further, Haykin identified three fundamentally different classes of network architectures [27]: the single-layer network, the multi-layer network as well as the recurrent network.

Single-layer Networks In a *layered* neural network, the neurons are organized in layers. The structure is a *directed, acyclic graph* and the signal is passed from one layer to another in a strict *feedforward* manner. An example is drawn in the figure 3.6. The single-layer network is composed of two layers, each consisting of a different type of nodes. The first class represent the input nodes illustrated as squares, which act as signal sources for the network. The second layer contains all the neurons, that return the transformed signals as output values. This layer is therefore called the *output layer*. Although the network actually consists of two layers, it is called a single-layer network. In the input nodes no computation is done, so their layer is omitted from the count. [27]

Multi-layer Networks Multi-layer networks have similar properties as single-layer networks. They too can be viewed as directed, acyclic graphs and their signal processing is again strictly feedforward. The difference lies in one or more *hidden layers*. The neurons in these layers are called *hidden neurons*. With additional layers of neurons, the network is able to work with more complex problems, for example extract higher-order statistics [27].

The input signal coming from the source nodes is passed to the *first hidden layer* of neurons. They process the signal and the output is then used as input for the second hidden layer or the output neurons, depending on the structure. Figure 3.7 illustrates an artificial neural multi-layer network with one layer of hidden neurons. [27]

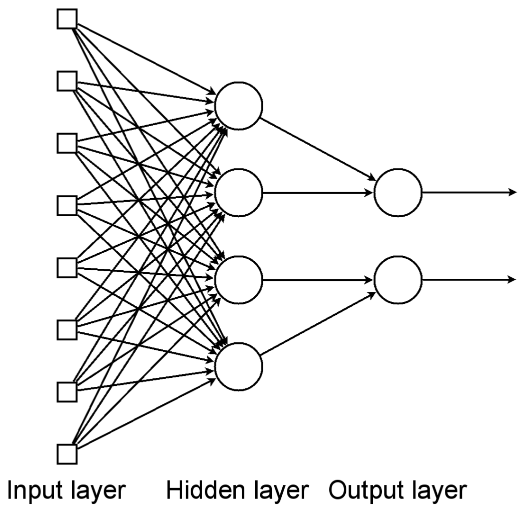


Figure 3.7: An example of a partially connected multi-layer neural network. (based on [27])

Recurrent Networks A *recurrent neural network* is significant different from the two classes already presented. The signal process of a recurrent network is not strictly feedforward, since it has at least one *feedback loop*. The use of feedback loops in a neural network has a great impact on the learning capability and performance of the network.

Feedback occurs in nearly every part of the nervous system of every animal. In general, it describes the functionality of a dynamic system, that is able to use the output of an element to influence the input of the same element. In the special case of a directed neural network, feedback is an arc from the output of a neuron to its input or to a preceding layer, building a cycle. The target of the arc on one of the preceding layers can be the input of one or more neurons. [27]

For example, a recurrent network may consist of a layer of input nodes fully connected to a layer of neurons. The neurons feed their output to some of the input nodes, generating a feedback loop. Some of the neurons act as output neurons, leaving the remaining neurons as hidden neurons. A possible recurrent neural network with hidden layers is drawn in figure 3.8.

3.2.4 Learning in Artificial Neural Networks

One of the most important abilities of a neural network is the ability to *learn* from its environment and adapt its actions towards the environment. It is not possible to find a globally valid definition of the term ”learning”, since there exist too many different facets of the activity learning. However, if we stick to the topic of neural networks, we can use a definition of learning by Mendel and McLaren [54]:

> Learning is a process by which the free parameters of a neural network are adapted through a process of stimulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter changes take place. (Mendel [54])

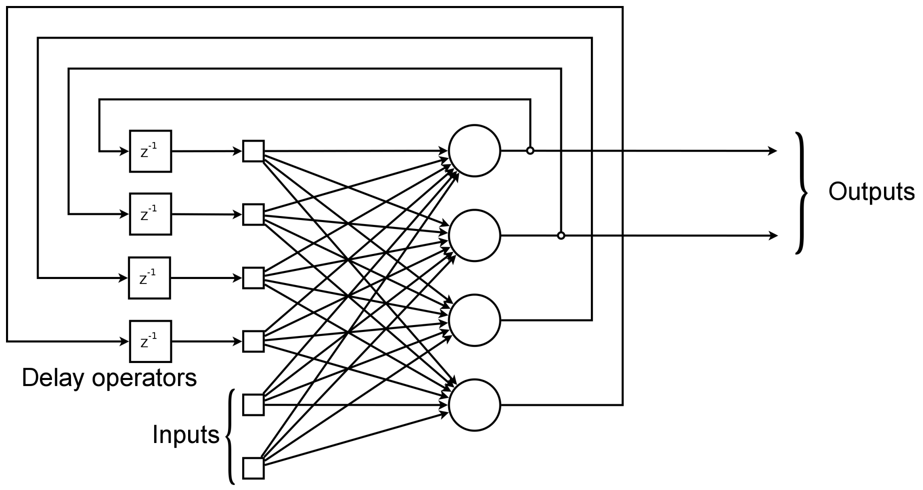


Figure 3.8: An example of a single-layer neural network with feedback loops. (based on [27])

This definition implies the following sequence of events that are required to be executed for a neural network to be able to learn [27]:

1. The neural network is *stimulated* by the environment.
2. The neural network react to this stimulations and undergoes *changes* in its free parameters.
3. With the changes made, the neural network responds to the environment in a *different* way.

The main goal of learning for neural networks is to improve the performance of the network. The improvement is done iteratively in many small steps, defined by some algorithm. A neural network learns about its environment through the information that the environment applies to the network. The stimulated network then adapts its synaptic weights and bias levels accordingly. Ideally the neural network can improve its performance with every iteration of the learning process.

The following sections will deal with examples of learning methods.

Error-Correction Learning The error-correction learning rule is based on the idea of optimization. The neurons are given a target to which they should iteratively adjust. Consider a single neuron k, driven by a input vector $x(n)$ coming from other neurons, which are themselves driven by an input vector applied to the source nodes. The natural number n represents a time step of an iterative training process. After an iterative step, the output y_k of the neuron is compared to a given *target* output $d_k(n)$. The *error signal* produced by these values is:

$$e_k(n) = d_k(n) - y_k(n) \tag{3.8}$$

The error signal $e_k(n)$ is used in an adjustment step that is applied to the synaptic weights of the neuron k. These adjustments are designed to iteratively make the output signal $y_k(n)$ come closer to the desired target output $d_k(n)$.

Hebbian Learning The oldest and most famous learning algorithm is named in honour after the neuropsychologist *Donald O. Hebb*. It is based on the descriptions of the behaviour of biological neural cells, which Hebb described in his book ”The Organization of Behaviour” [29]. The hebbian learning rule consists of two sub rules [27]:

- If two neurons on either side of a synapse are activated synchronously, then the strength of that synapse is selectively increased.
- If two neurons on either side of a synapse are activated asynchronously, then this synapse is selectively weakened or eliminated.

A synapse with this properties is called a *Hebbian synapse*.

Competitive Learning In a competitive learning method, the output neurons compete against each other, in order to become active. In a network based on, for example hebbian learning rules, it is possible that more than one neuron become active at a time. In a competitive environment only one neuron can be active at the same time. Haykin describes three basic preconditions for a competitive learning rule [27]:

- A set of equal neurons, initialised with random synaptic weights. Therefore every neuron will respond differently to a given set of input signals.
- A limit on the ”strength” of every neuron.
- A mechanism to limit the number of active neurons in a group to one at a time. The winning neuron is called the *winner-takes-all neuron*.

For a neuron k to be the winning neuron of its group, its output value v_k for a given input vector must be the largest among its group. The output signal y_k of the winning neuron is set to one, whereas the signal of all other neurons is set to zero. The learning is done by a shifting of synaptic weights. Every neuron shifts some amount of weight from the inactive synapses (with a value of zero) to its active neurons. Therefore the synapses of the winning neuron are strengthened, while the synapses of the loosing neurons are weakened.

3.3 Self-Organizing Maps

A special application of an artificial neural network is the so-called *Self-Organizing Map (SOM)*. These networks are based on the principle of *competitive learning*, where only one neuron can be active at a time [27]. This learning method was explained earlier in section 3.2.4. The neurons are arranged in a two-dimensional array and are connected to a defined *neighbourhood set* of nearby neurons [42].

The ideas for self-organizing maps are based on the knowledge gained about the structure of the human brain. The various areas of the brain and especially the cerebral cortex are organized according to different sensory inputs. These areas perform specialized tasks, for example for processing tactile, visual and acoustic sensory input. Every specialized region is again divided in several subregions. These fine-structured areas are organized according to the topographical origin of the response signal, e.g., the somatic sensory areas for hand and fingers all lie close together on the somatosensory cortex of the brain. This configuration is useful in many ways. By bringing similar brain functions close together, the wiring and the ”crosstalk” between functions can be minimized and the brain becomes more efficient, more logical and more robust. [42]

This knowledge about the self-organizing representation of information that the brain is capable of, led to the development of a uniform, singly-connected, one-level medium that is able to represent *hierarchically related data*. This type of medium is known as *Self-Organizing Maps*.

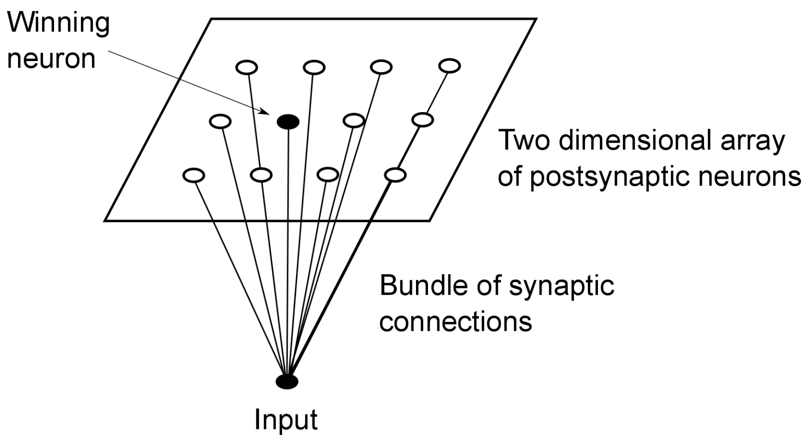

Winning neuron
Two dimensional array of postsynaptic neurons
Bundle of synaptic connections
Input

Figure 3.9: A SOM with synaptic connections and the winning neuron. [27]

Different areas of the map are assigned different abstraction levels of information and represent a tree structure. This tree structure can be used to extract taxonomic and cluster information. [42]

As mentioned earlier, a SOM is a artificial neural network, whose neurons are usually structured in a regular two-dimensional grid. Higher-dimensional maps would be possible, but are not as common [27]. Self-organizing maps may be described formally as a non-linear, ordered, smooth mapping of high-dimensional input data mapped onto the elements of a regular, low-dimensional array [42].

3.3.1 Learning in Self-Organizing Maps

Let m denote the dimension of the input data and l denote the total number of neurons in the network. Let x be an m-dimensional input vector (adapted from [27]):

$$x = (x_1, x_2, \dots, x_m) \tag{3.9}$$

Each neuron is represented as a m-dimensional synaptic weight vector:

$$w_j = (w_{j1}, w_{j2}, \dots, w_{jm}),\ j = 1, 2, \dots, l \tag{3.10}$$

The neurons are initialised with random values, preferably the values are taken from the same domain as the input samples. We require a general distance metric between x and w_j denoted as $d(x, w_j)$ [42]. The distance metric may be the Euclidean distance measure [42], defined as:

$$d(x, w_j) = \sqrt{(x_1 - w_{j1})^2 + (x_2 - w_{j2})^2 + \dots + (x_m - w_{jm})^2} \tag{3.11}$$

When an input vector is applied to the network, the competitive learning rule dictates that only one neuron can be active, winning the competition. The winning neuron is also called the *image* of the input vector x on the SOM array. It is defined as the array element w_c that matches best with x where (adapted from [42]):

$$c = \arg\min_j \{d(x, w_j)\},\ j = 1, 2, \dots, l \tag{3.12}$$

The situation, where an input vector is applied to the neural network is illustrated in figure 3.9. One neuron lies closest to the input vector and is therefore the winner of the competition.

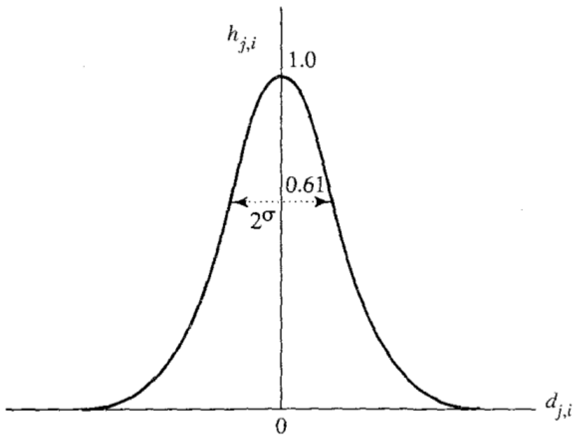


Figure 3.10: A graphical interpretation of the Gaussian neighbourhood function. [27]

The winning neuron and its *geometric neighbourhood* are stimulated by the input vector x. Thus, they will try to decrease the distance of their synaptic weight vectors in regard of the position of x. This continued shifting of positions of the neuron vectors leads to a *global ordering*. The learning process is defined as [42] :

$$w_j(t+1) = w_j(t) + h_{cj}(t)[x(t) - w_j(t)] \tag{3.13}$$

Where $t = 0, 1, 2, \ldots$ is a discrete time coordinate, numbering the steps of the learning phase. A great impact on the learning process has the *neighbourhood function* $h_{cj}(t)$, a smoothing kernel function over the array of nodes. Based on the position of the winning neuron i, it defines the set of nodes that are affected by the stimulation. The function has its maximum at the winning neuron ($d_{i,i} = 0$), monotonically decreases with increasing distance ($d_{i,j}$) to neuron i and decaying to zero for $d_{i,j} \to \infty$ [27]. A widely applied neighbourhood kernel is the *Gaussian function*:

$$h_{i,j} = \alpha(t) \cdot exp\left(-\frac{d_{i,j}^2}{2\sigma^2}\right) \tag{3.14}$$

The parameter σ controls the shape of the graph and thus defines the topological width of the neighbourhood. An example of a Gaussian neighbourhood function is illustrated in figure 3.10. The function $\alpha(t)$ is a monotonically decreasing function and represents a *learning-rate factor*. It is important for convergence reasons, since with increasing time, the neighbourhood selected for stimulation decreases in size and the learning process slowly decays. [42]

3.3.2 Example

An example of the training of a SOM is illustrated in figure 3.11 (from [42]). The training data is uniform distributed, as seen in subfigure 3.11(a). Subfigure 3.11(b) illustrates the first step of the training, at the time when the synaptic weight vectors $w_j(0)$ of the neurons are initialised with random values from the same domain as the training data. Then the training is started and the network unfolds into a *mesh* and the structure becomes visible after some time (subfigure 3.11(c)). After the learning process died off due to convergence, the geometric structure of the network is approximately uniform distributed like the input data, as seen in subfigure 3.11(d).

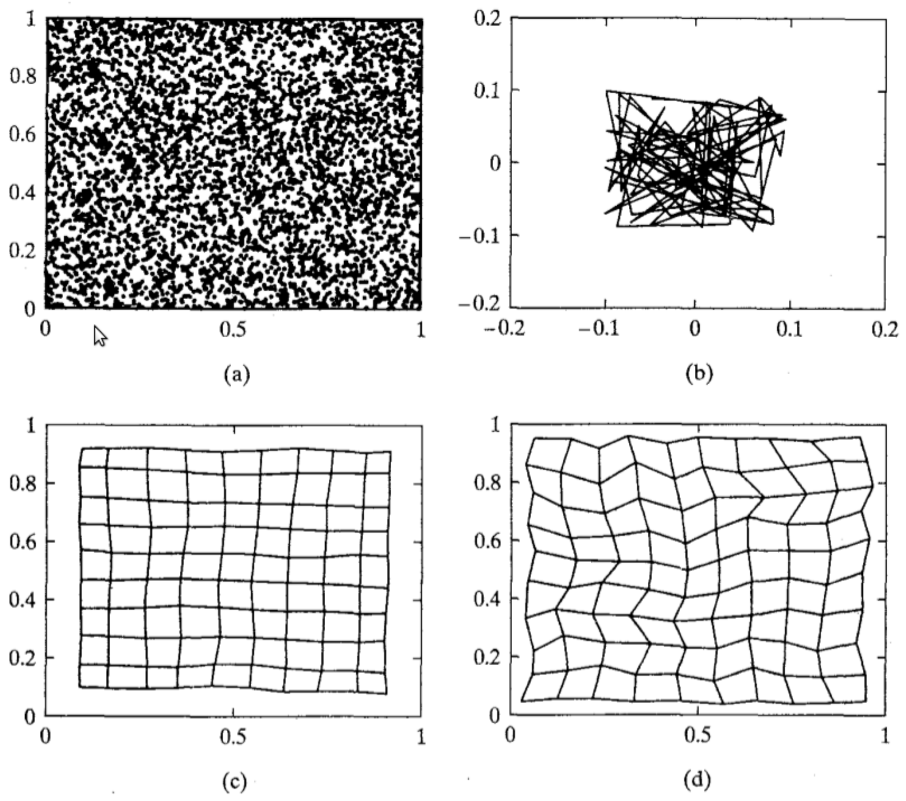


Figure 3.11: Stages of a SOM training process: (a) the distribution of the training data; (b) the SOM is initialised with random data; (c) during the training of the SOM; (d) training is finished, the maps now spans over the entire dataset, representing its distribution. [42]

3.4 Support Vector Machines

One classification approach that has become popular in recent years is the *Support Vector Machine (SVM)*. The concept behind SVM is to find an optimal hyperplane that separates the data in two sectors, according to their classes. Basically it is a linear classification method but can be extended to separate non linear separable data [3][72] and even to find clusters for one-class problems [2][3].

Traditionally SVMs use the labels $y = -1$ and $y = +1$ for the two classes C_1 respectively C_2. Consider the m-dimensional training samples x_i with $i = 1, \dots, n$. If the training data is linearly separable, a decision function D can be defined:

$$D(x) = w^T x + b \tag{3.15}$$

where w is an m-dimensional vector and b is a bias term. For $i = 1, \dots, n$

$$w^T x_i + b \begin{cases} > 0 & \text{for} \quad y_i = +1 \\ < 0 & \text{for} \quad y_i = -1 \end{cases} \tag{3.16}$$

Since we required the training data to be linearly separably, no data point satisfies $w^T x + b = 0$. Therefore it is sufficient to consider the following inequalities:

$$w^T x_i + b \begin{cases} \geq +1 & \text{for} \quad y_i = +1 \\ \leq -1 & \text{for} \quad y_i = -1 \end{cases} \tag{3.17}$$

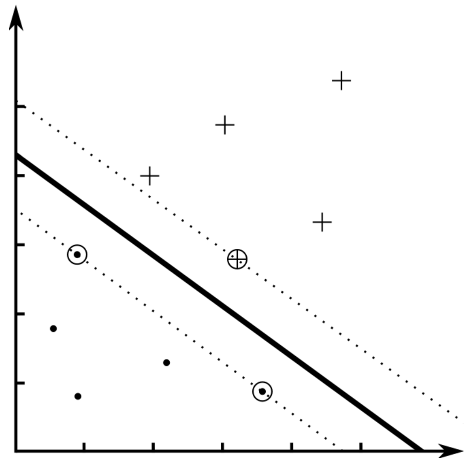


Figure 3.12: A SVM example showing an optimal separating hyperplane. [3]

which is equivalent to

$$y_i(w^T x_i + b) \geq 1 \quad \text{for} \quad i = 1, \ldots, m \tag{3.18}$$

It is intended that the equation be $\geq +1$. We require that the instances should not only be on the correct side of the hyperplane, they should even have some distance to it. The distance from the hyperplane to the closest instance is called margin. Maximizing the margin allows the method to correctly classify test instances that were slightly displaced by noise interferences. The hyperplane with the largest margin is called *optimal separating hyperplane*. [2] [3]

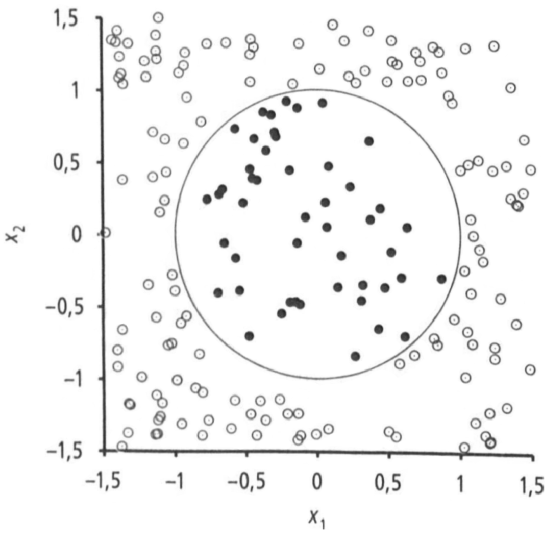
An example is drawn in figure 6.5. The two classes are shown by dot and plus signs, the thick line is the hyperplane and the dashed lines define the margin. The support vectors of the margin are drawn in circles.

So far we assumed the training data to be linearly separable. This is a valid assumption, but often the training data will not meet this requirement and the linear classifier may not perform as well as expected. A solution to this problem is to map the training data into a high-dimensional space and use the linear classifier in this new space. [2]

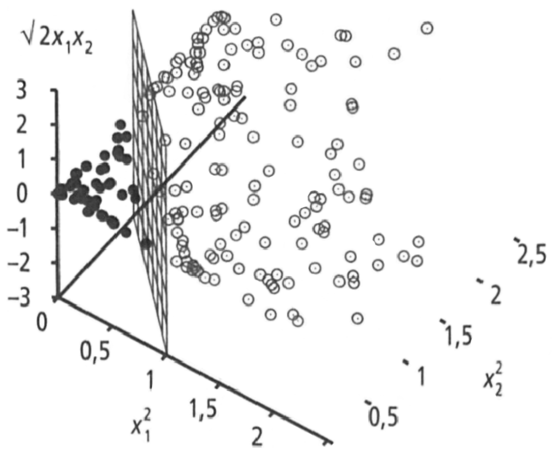
An example of non linearly separable training data is drawn in figure 3.13a. The two classes are drawn as white and black circle. By mapping the same training data into a higher dimension, it becomes linearly separable. Figure 3.13b shows the same training data mapped into three dimensions. The possibility of separation is indicated by the hyperplane. [72]

To determine the optimal separating hyperplane, we need to find a w with the minimum Euclidean norm that satisfies equation 3.18. A solution to this problem can be obtained by solving the following optimization problem for the two variables w and b [3]:

$$\begin{aligned} &\text{minimize} && Q(w,b) = \tfrac{1}{2}\|w\|^2 \\ &\text{subject to} && y_i(w^T x_i + b) \geq 1 \quad \text{for } i = 1, \ldots, m \end{aligned} \tag{3.19}$$



(a) The training set in two dimensions.



(b) The training set mapped into three dimensions.

Figure 3.13: A training set with a separating hyperplane in two and three dimensions. [72]

Due to the square of the Euclidean norm $||w||$ this problem becomes a quadratic optimization problem. In addition, this problem can further be converted into an equivalent problem whose number of variables is only the number of training data [3]:

$$\begin{aligned} &\text{minimize} \quad Q(\alpha) = \sum_{i=1}^{m} \alpha_i - \tfrac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j x_i^T x_j \\ &\text{subject to} \quad \sum_{i=1}^{m} y_i \alpha_i = 0, \quad \alpha_i > 0 \quad \text{for } i = 1, \ldots, m \end{aligned} \tag{3.20}$$

The details of this conversion, however, are not part of this thesis and may be looked up in several technical books, including the book ”Introduction to Machine Learning” by Ethem Alpaydın [3].

3.5 Clustering

The unsupervised learning method of *fuzzy clustering* is based on ideas from the field of *fuzzy set theory*. Fuzzy set theory on the other hand is an extension of the classical set theory, which is sometimes called *crisp set theory* in this context. The classical theory is build on a fundamental concept of a ”set” where an individual either is a member or is no member of this set. There exists a crisp and unambiguous distinction between a member and a nonmember. Mathematically speaking, let A be a nonempty set. To indicate that an individual x belongs to the set A we write [14]:

$$x \in A \tag{3.21}$$

If x is not a member of A we write:

$$x \notin A \tag{3.22}$$

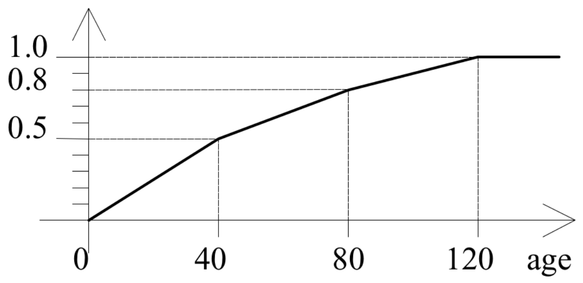


Figure 3.14: An example of a membership function for fuzzy sets. [14]

Thus, we can define a *characteristic function* χ of A [14]:

$$\chi_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases} \tag{3.23}$$

On the contrary to this sharp distinction of the classical set theory, the fuzzy set theory accepts partial membership of a set. In a way it thereby generalizes the classical set theory to some extent [14].

In order to allow an element a partial membership of a set, we need to generalize the characteristic function. This is done by describing a membership grade of the elements in the set: larger values describe higher degrees of membership. For example, consider the set S containing all humans and the subset S_f which is defined by

$$S_f = \{s \in S | s \text{ is old } \} \tag{3.24}$$

The subset S_f can not be defined by classical set theory, since the property ”old” is not well defined and can not be precisely measured. Therefor, in order to define S_f we have to quantify and define the concept ”old”. One way to describe the concept is by a curve shown in figure 3.14, where the only people who are considered to be ”absolutely old” are those who are 120 years old or older. The ones who are considered to be ”absolutely young” are the newborns. All other people are considered old and young at the same time. A person who is 40 years old is equally old and young with a degree of 0.5 for both. [14]

The curve illustrated in figure 3.14 is in fact a generalization of the classical characteristics function χ_{S_f} and is called a *membership function* of subset S_f. By this function it can be determined if a person is or is not a member of S_f. We will denote this function by $\mu_{S_f}(s)$ with $s \in S$.

The subset S_f together with the membership function μ_{S_f} is called a *fuzzy subset*. A fuzzy set always consists of these two components: a set and a membership function. This is in contrast to the classical set theory, where every set uses the same two-valued characteristic function, as previously described in formula 3.23. [14]

3.5.1 Hard c-Means

c-Means is a clustering method based on classical set theory. Each data point will be assigned to exactly one data cluster, sometimes called *partitions*. A valid partition of all data points has to meet several requirements: The data points form the set $X = \{x_1, x_2, \ldots, x_n\}$. A partition U consists of several sets $U := \{A_i, i = 1, 2, \ldots, c\}$. The union of the sets have to contain every data points, every data point has to be in a set (formula 3.25). No sets may overlap, every data point can be in at most one set (forumla 3.26). No set may be empty or contain all data points (formula 3.27). [71]

$$\bigcup_{i=1}^{c} A_i = X \tag{3.25}$$

$$A_i \cap A_j = \emptyset \quad \forall i \neq j \tag{3.26}$$

$$\emptyset \subset A_I \subset X \tag{3.27}$$

The relationship between data points and classed can be illustrated as matrices, the so-called *partition matrices*. The columns correspond to the data points x_i, the rows to the partitions A_1 and A_2. Some examples are [71]:

$$U_1 = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad U_2 = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.28}$$

By this definition, it is obvious that there are several partitions of a given set of data samples possible. The task of the clustering method is to select the most reasonable c-partition U of all possible partitions, denoted as the set M_c. To solve this task, an objective function that will be used to cluster the data is required. One proposed algorithm is known as a within-class sum of squared errors approach, using a Euclidean norm to characterize distance. The function that is used by this algorithm is denoted $J(U, v)$ and is defined as [71]:

$$J(U, v) = \sum_{k=1}^{n} \sum_{i=1}^{c} \chi_{A_i}(x_k) d(x_k, v_i)^2 \tag{3.29}$$

Where U is a partition matrix and the parameter v is a vector of cluster centers. The Euclidean distance measure is $d(x_k, v_i)$ (see formula 3.11) and determines the distance between each data point x_k and its cluster center v_i. The characteristics function is $\chi_{A_i}(x_k)$ (see formula 3.23). The clustering method seeks the optimum partition U^*, which is the partition that gives the minimum value for the function J [71]:

$$J(U^*, v^*) = \min_{U \in M_c} J(U, v) \tag{3.30}$$

Finding the optimal partition can become a very difficult task for practical problems, because $M_c \to \infty$ for even modest-size problems. An exhaustive search for optimality is obviously not computationally feasible, but effective alternative search algorithms have been designed. One such algorithm is known as *iterative optimization* and was introduced by J.C. Bezdek [9]. [71]

3.5.2 Fuzzy c-Means

The Fuzzy c-Means algorithm works similar to the algorithm of Hard c-Means. As explained earlier, fuzzy set theory allows individuals a partial membership in more than one set. Applied to the field of clustering, a data sample can have a partial membership in more than one class or cluster. [71]

First of all, we have to define a set of fuzzy sets $\tilde{U} := \{A_i, i = 1, \ldots, c\}$ that form the c-partition of all data points X. The characteristics function χ needs to be extended to an membership function

μ, displaying the degree of membership of a data point x_k to a class A_i [71]:

$$\mu_{A_i}(x_k) \in [0,1] \tag{3.31}$$

A valid partition in fuzzy clustering has to satisfy similar constraints as in crisp clustering. First of all, the sum of all membership values of a data point has to be unity, equivalent to the crisp case. Formally written as

$$\sum_{i=1}^{c} \mu_{A_i}(x_k) \quad \forall k = 1,2,\ldots,n \tag{3.32}$$

The restriction of formula 3.26, that every data point can be in at most one class, can be dropped. In fuzzy set theory, each data point can have multiple partial memberships. The expression of formula 3.27, stating that no set can be empty or contain all data points, remains in effect, though. [71]

In contrast to Hard c-Means the partition matrices can contain every number $\in [0,1]$. An exemplary partition matrix $\tilde{U}$ is:

$$\tilde{U} = \begin{bmatrix} 0.91 & 0.58 & 0.13 \\ 0.09 & 0.42 & 0.87 \end{bmatrix} \tag{3.33}$$

Again, the columns of the matrix represent a data samples x_k, whereas the rows stand for two classes A_1 and A_2. In this case, x_1 has membership values of 0.91 for class A_1 and 0.09 for class A_2. [71]

To determine the optimal c-partition $\tilde{U}$ for grouping a collection of n data points into c classes, we require a fuzzy objective function J_m:

$$J_m(\tilde{U}, v) = \sum_{k=1}^{n} \sum_{i=1}^{c} \mu_{A_i}(x_k)^{m'} d(x_k, v_i)^2 \tag{3.34}$$

As distance metric again the Euclidean distance $d(x_k, v_i)$ between the ith cluster center and the kth data point is used. The difference to Hard c-Means, which used a characteristic function χ, is that the fuzzy objective function uses the membership function μ defining the degree of membership between data point x_k and class A_i. A new parameter m' is introduced, which is called a *weighting parameter*. It has a range of $m' = [1, \infty)$ and is used to control the amount of fuzziness in the clustering. [71]

The optimum fuzzy c-partition will be the smallest of the partitions described by formula 3.34:

$$J_m(\tilde{U}^*, v^*) = \min_{\tilde{U} \in M_{fc}} J(\tilde{U}, v) \tag{3.35}$$

The problem of exploding complexity with larger instances is even further tightened, since the number of possible, valid c-partition is now infinite. As with many optimization problems, however, the solution to formula 3.35 cannot be guaranteed to be the best solution (the *global* optimum). Thus, the best solution given by a predefined level of accuracy (e.g., a *local* optimum) will suffice. The *iterative optimization* algorithm by Bezdek [9] is again an effective way of finding an adequate solution to the fuzzy c-clustering problem. [71]

3.6 State of the Art

Using artificial intelligence method to enhance security system is a well studied topic. Machine Learning concepts like the four presented are often used to learn the normal system behaviour and detect abnormal behaviour during runtime.

Jalil *et al.* [36] compare three ML methods (Artificial Neural Network (ANN), SVM and decision tree) for their use in IDS systems. Firdausi *et al.* [22] presented an analysis of ML techniques used in an behaviour based malware detection system. The classifiers compared in their work are: k-Nearest Neighbours, Naive Bayes, J48 Decision Tree, SVM and Multilayer Perceptron Neural Network.

3.6.1 Artificial Neural Network

A well studied topic is the use of an ANN for classification of a behaviour of a system, e.g., network traffic. Several publications exist that propose or improve ANNs for the use within an IDS, for example Gosh *et al.* [25], Han *et al.* [26] or Zhao *et al.* [90]. Linda *et al.* [47] investigate the monitoring of network timing behaviour for its use within an IDS to detect abnormal network behaviour. Shun and Malki [77] presented an approach with a so-called feedforward neural network and trained it with the Back Propagation (BP) algorithm.

Another interesting combination showed Jiang and Zhao [37] and similar Zhou and Yang [91]: They combined a Genetic Algorithm (GA) with a BP Neural Network to gain the good global searching ability of genetic algorithms with the accurate local searching feature of the neural network.

A Multi-Agent System in combination with a neural network classifier has been used in the paper of Júnior *et al.* [39]. The agents search for binaries on a file system and let the neural network decide whether the binary is vulnerable or not. They can communicate with each other and can request the help of nearby agents to work on a specific region of the workspace. The neural network is specialised in heap- and stack-overflows, but due to its learning capabilities, it can detect other classes of vulnerability, such as format string, integer overflows and others.

The applicability of neural networks in automated malware detection investigated Gavrilut *et al.* [15]. They analysed two variants of networks: a cascaded one-sided perceptron and cascaded kernelized one-sided perceptrons.

3.6.2 Self-Organizing Map

A Self-Organizing Map is used by Ramadas *et al.* [69] to create an anomalous network traffic detector. The SOM is trained by a set of sample web connections. To make sure the training set does not contain intrusions itself, the data was checked by a rule based intrusion detection system at first. After a two phased training and a final validation phase, the SOM was tested and produced promising results.

Kiziloren and Germen [41] introduced a SOM-based approach to classify traffic in a network in three classes: port scanning, heavy-downloading and the rest. In their test setup they achieve a success rate of near certainty.

Ting *et al.* [84] address the robustness and reliability problem of some SOMs by replacing the distance metric by a non-linear kernel function, gaining higher classification precision.

Intrusion detection is an often used appllicaton for Self-Organizing Maps, for example Höglund *et al.* [31], Lichodzijewski *et al.* [46] and Ippoliti and Zhou [35] successfully created an IDS based on a SOM.

Powers and He [66] presented a hybrid approach for Intrusion Detection Systems. The initial detection of an anomalous network connection is done by an artificial immune system. This suspicious connections are then further classified by a SOM. The SOM allows the extraction of higher-level information like membership to a specific cluster and with this, the attack type.

3.6.3 Support Vector Machine

In an evaluation work of Mukkamala *et al.* [56] they compared an Intrusion Detection System based on neural networks and an IDS based on SVMs. Both systems were trained with the same data set originating from a competition designed by the Defense Advanced Research Projects Agency (DARPA). Both systems showed a result greater than 99% accuracy. The authors predicted a great potential for the SVM, due to its scalability and faster training and running time.

A similar conclusion reaches Ambwani [4] within the work on a Multi-Class SVM for intrusion detection. A one-versus-one approach with a Radial Basis Function (RBF) kernel was trained and tested on a huge sample set of seven million connection records. The sample data was again taken form a contest created by the DARPA. Further, Bao *et al.* [7] present a combination of SVM algorithms with Anomaly Intrusion Detection and Misuse Intrusion Detection in their work.

Khan *et al.* [40] use a SVM for detecting network-based anomalies and address the problem of a long training time of the SVM with the use of a special clustering algorithm, called Dynamically Growing Self-Organizing Tree. The same problem is addressed by Mulay *et al.* [57], they propose the use of a decision tree for preparing the training data.

A possible problem for SVMs is that they are sensitive to noise in the training sample. The presence of mislabelled data can lead to poor generalization ability and classification accuracy. Hu *et al.* [32] address this problem by using a so-called Robust Support Vector Machine (RSVM). This extension to regular SVMs uses an adaptive margin as well as an ”average” algorithm [80]. With this method, a particular sample in the training set only contributes little to the final result. The effect of outliers can be eliminated by taking averages on the samples.

Rieck *et al.* [70] presented an study on SVMs for their applicability in automatic behaviour based malware detection.

3.6.4 Clustering

An intrusion detection system based on clustering was presented by several authors, for example Portnoy [65] and Oh *et al.* [62]. The approach of Oh reads a data stream, which is an ordered sequence of objects, and clusters these objects for anomaly detection. In conventional methods, the number of clusters is given in advance. This can lead to an identification of inaccurate clusters. The proposed algorithm puts every object in its own cluster until a given number of clusters is reached. Every later object is assigned to one of the existing clusters. To maintain the quality of the clusters they can be split up or merged if necessary.

In another work of the same authors [61] they modify their clustering algorithm to build a behaviour profile of a user. The method extracts several features of an audit data log and generates a long-term profile containing a statistical summary for each feature. To detect an anomaly, the activities of the user are summarized into a short-term profile and compared with the long-term profile. If the difference between two profiles is large enough, the on-line activities are considered as anomalous behavior. A similar approach was presented by Park *et al.* [63].

A clustering algorithm based on fuzzy c-means is introduced by Hubballi *et al.* [33]. They identify a huge problem of IDSs in the large amount of false positives and false negatives caused by outliers. The proposed algorithm can control the degree of false alarms, by adjusting the threshold of membership. This leads to a higher accuracy and detection rate compared to related works.

Jiang *et al.* [38] proposed a method to detect outliers by a two-phase clustering process. In the first phase, they modify the traditional c-means algorithm by using a heuristic and build a minimum spanning tree in the second phase.

4 Architecture of the Proof of Concept Implementation

The previous chapters explained the topics of IT Security, Software Testing and Artificial Intelligence to some extent. These chapters build the foundation of the following pages where Security Testing and Artificial Intelligence are combined.

The basic concept of the combination is based on learning the normal behaviour of the tested system. If the security testing tool knows how the system works under normal circumstances, then it can also automatically detect changes in the behaviour during the test and tell the executing tester which input data called on this behaviour change. The tester can then investigate this situation and draw conclusions to possible vulnerabilities in the system.

The first section presents an architecture of security testing frameworks, which will later be used in the proof-of-concept implementation. The architecture of the selected fuzzing software is explained in section 4.2 and is subsequently extended with artificial intelligence modules. Some prototype implementations of the security framework extended with various AI methods are then evaluated in chapter 6.

4.1 Architecture of Security Testing Frameworks

In general, a security test setup consists of two systems: the *security test system* and the *test subject* (i.e., System Under Test, SUT). As any human tester would do, the test system sends requests to the SUT and analyses the responses. It checks the responses against some predefined specifications to decide whether the system has *passed* or *failed* the tests. The following description of the architecture is based on the originating paper of this thesis called ”Generic Approach for Security Error Detection Based on Learned System Behavior Models for Automated Security Tests” [73].

Figure 4.1 shows a test setup with a security test system and a System Under Test. The security test system is positioned on the top and the SUT is on the bottom. The figure shows a simplified architecture of the components of both systems.

The SUT contains various components which communicate with internal communication interfaces. Often the tested system has a user interface attached. The operating system and the underlying hardware are part of the system too. Additionally, the SUT communicates with external components which are not part of the test target, for example a database server.

The security testing framework contains the test execution controller as central component. It manages the execution of learning and security test cases. In the module ”Preparation of Test Data”, two engines generate and prepare the messages that are sent as requests to the SUT. One each for the learning and for the test cases. ”Learning and Determination” contains all components that are responsible for analyzing the SUT behaviour and drawing conclusions from the gained data.

Each *analyzer* observes one specific behaviour and deduces a metric. To combine the single measured values from the analyzers, the *analyze manager* is used. It collects the results of all analyzers and performs the learning and clustering. The manager triggers the single analyzer

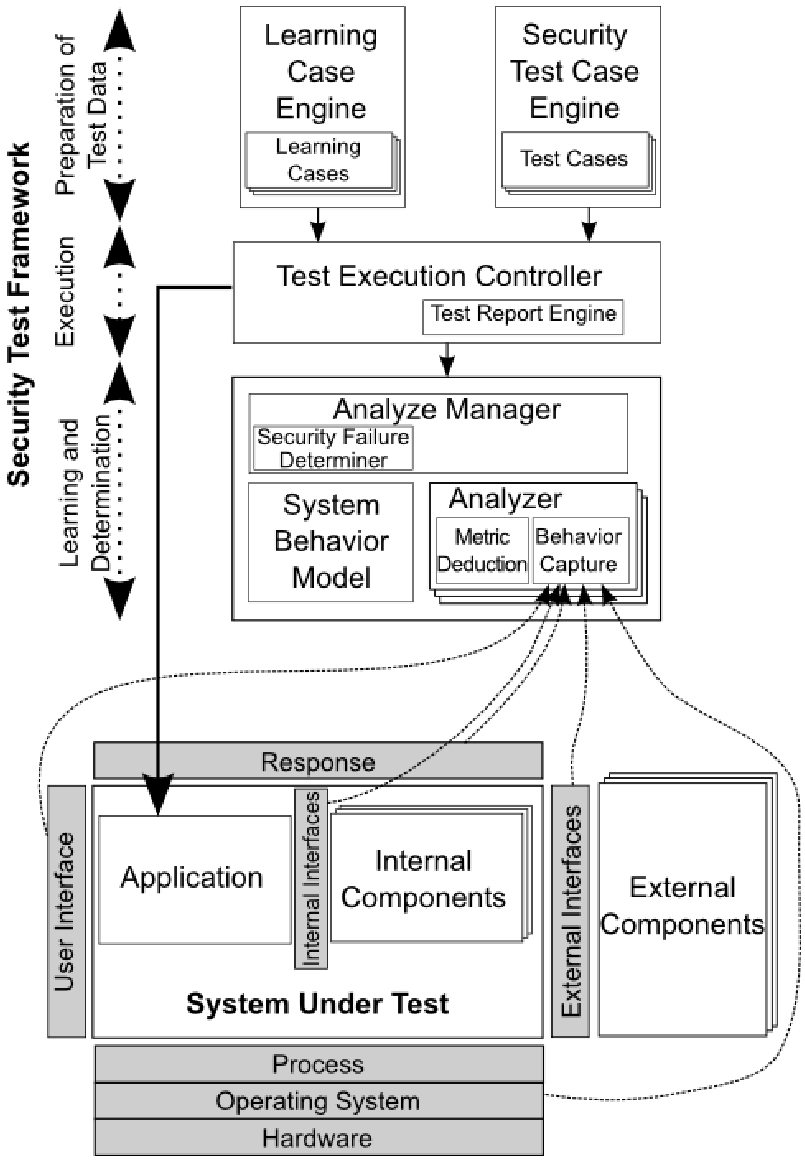


Figure 4.1: Architecture for learning system behaviour and determining security failures. [73]

before and after the execution of the security test/learning case. Before the security test/learning case the analyzers are informed, that they should start capturing the system behaviour. After the security test/learning case the analyze manager collects the determined metrics for the security test cases and for learning cases from the single analyzers. The metrics from all analyzers are combined by the analyze manager to a single behaviour. Based on this observed behaviour and changes in the behaviour, the analyze manager decides whether a security failure is present or not. A more detailed description of the functionality of the components in the ”Learning and Determination” module can be found in section 4.3.2

4.2 The Fuzzing Framework ”Fuzzolution”

The fuzzolution project is a software which provides a generic and easily expandable framework for security tests using the technique of fuzz testing (see section 2.3). The framework is based on the architecture of security testing frameworks described in the previous section.

4.2.1 Process of one Fuzzing Test Run

One *test run* is executed on one functionality of an application that requires arguments, for example an URL with various parameters. During a run, negative tests are sent in order to try and break the system. Each run repeatedly starts a *test iteration* until all predefined attack vectors have been selected. Basically, one iteration of the fuzzer consists of multiple steps:

1. select an *attack vector*
2. build the *attack string*
3. send the *request* to the SUT
4. run the *analyzers* to extract the metrics

The attack vector is an argument that is injected into the SUT in order to generate a response. The attack vector could either be automatically generated data or a string selected from a well-known attack list. An example SUT could be a simple web interface backed by a SQL database. The application expects an id of a user as input and displays the stored information. Possible attack vectors are:

```
<>"'%;)(&+
' or 1=1
1 or sleep(2000)#
```

Listing 4.1: Some examples of attack vectors.

The selected attack vector is then inserted into the *attack template*, which is defined by the configuration. The following is an example of a attack template taken from the configuration for one application of the Web Application Vulnerability Scanner Evaluation Project (WAVSEP) tool (see section 5.2 for a description of this tool). The substring `@@username@@` is replaced by the chosen attack vector and the whole attack string is send as Hyper Text Transfer Protocol (HTTP) request to the SUT. For readability reasons it is split up:

```
http://localhost:8080/wavsep/active/SInjection-
   Detection-Evaluation-GET-200Valid/Case03-
   InjectionInCalc-String-BooleanExploit-
   WithDifferent200Responses.jsp?username=@@username@@
```

Listing 4.2: An attack template for an application of the WAVSEP platform.

After the SUT has send a response, the configured analyzers are called to collect data as described in the previous section 4.1.

4.2.2 Composition of the fuzzolution tool

In order to achieve the flexibility required for security testing of various applications, the fuzzer is designed as a modular system. A framework defines the properties and functionality of every module in order to be able to work with the other modules. Around this framework a core installation includes implementations of all modules. These generic modules can be used in many test setups without the need of further development. It is enough to modify the configuration files

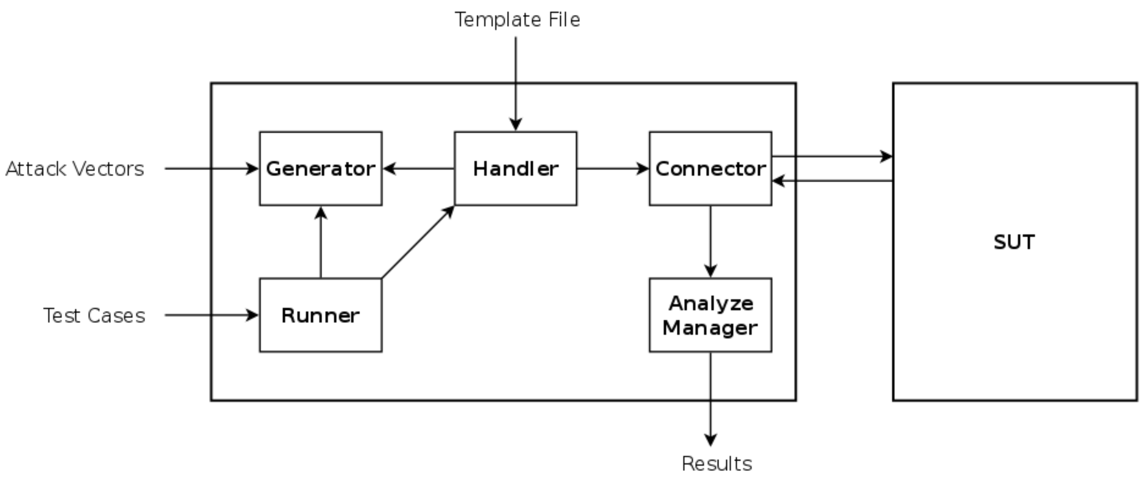


Figure 4.2: Architecture of the fuzzolution framework. (based on [74])

of every module to set up the fuzzer for the current testing project. However, if a SUT requires a module to have specialised features, it can easily be extended or replaced by a specific suiting implementation.

These mentioned base modules are called: Runner, Connector, Handler, Generator and Analyze Manager. The architecture of the fuzzolution framework is presented in figure 4.2. The image displays the command chain from Runner to the Generator and Handler. The Handler requests data from the Generator and starts the Connector. The Connector communicates with the SUT and calls the Analyze Manager afterwards.

Figure 4.3 represent an Unified Modeling Language (UML) sequence diagram of the activities inside the fuzzer. Visible is the initialisation sequence started by the Runner and relayed by the Analyze Manager to its Analyzers. The process of one test run, as described in words above, is displayed in the execution sequence. The Runner activates the Handler, which calls the Generator and later the Connector. The SUT response analyze sequence is again initialised by the Runner by starting the Analyze Manager. The Manager calls all his assigned Analyzers in sequential order:

In the following paragraphs all components are described with respect to their common core implementations.

Runner The *Runner* is the outermost module and acts as controller for the test execution. The configuration files for the tests are parsed by the entrance method and the contained information is provided to the Runner class. The Runner analyses the given information and iterates through all configured test cases. For every test case it activates the *Generators* in order to prepare the test case for execution. The *Handler* is called, taking the information gathered from the Generator and combines them with a predefined template file, completing the test case preparations. The actual test execution is done by the *Connector*, which is again called by the Handler class. The response of the SUT is eventually parsed by the *Analyze Manager*.

Generators The generator classes are responsible for the selection and generation of attack vectors. Every generator has a specific task of creating one kind of attack vectors. An example is a generator that produces random integer numbers in a defined range. A different type of generator reads input strings from a file containing attack vectors. In a test run several generators can be used and their combined output is the list of all attack vectors that will be used.

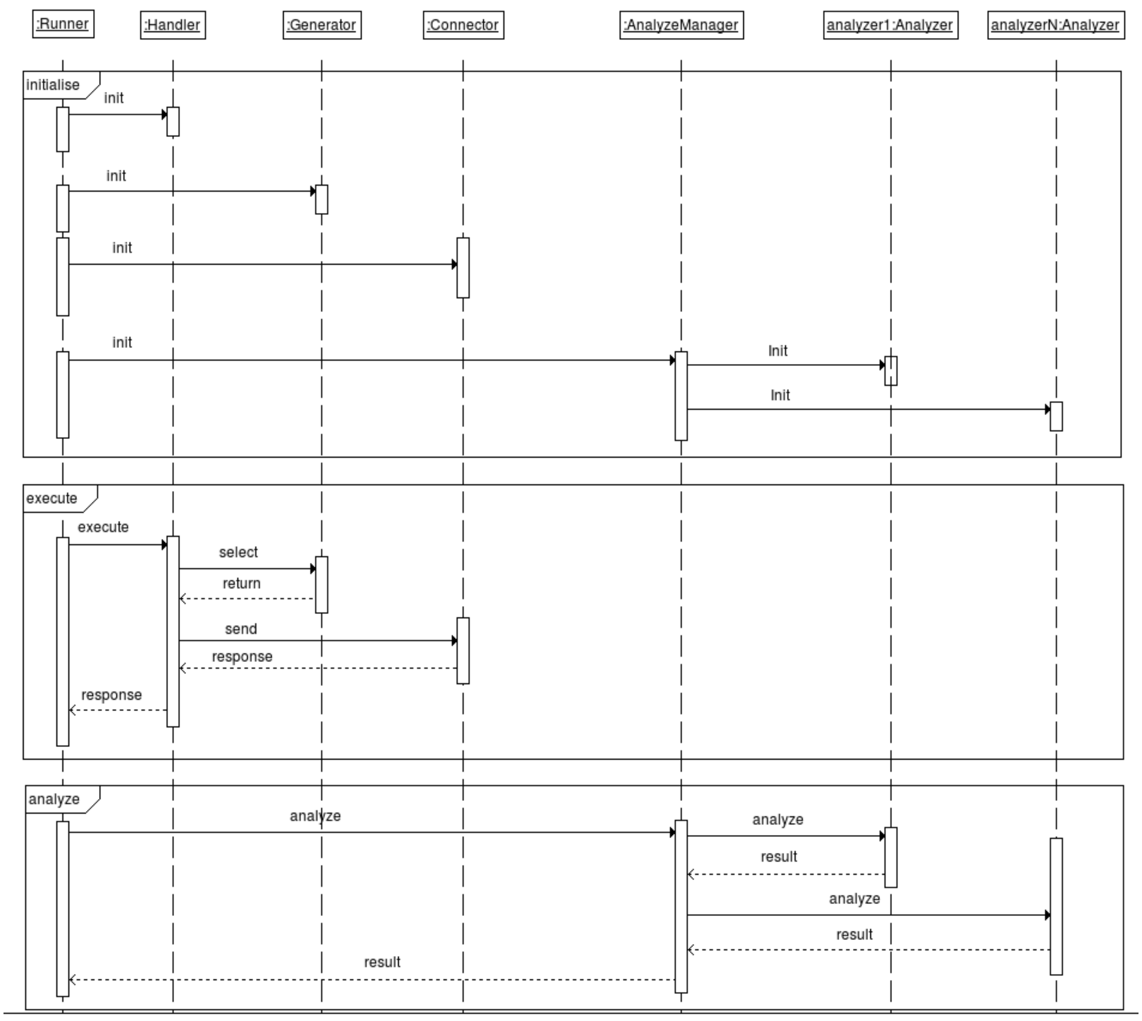


Figure 4.3: Sequence diagram describing the test execution of the fuzzolution tool.

Handler The handler module requires two arguments: the list of attack vectors produced by the generators and an attack template. For every argument defined in the template (e.g., `@@username@@`) and for every attack vector, the handler builds one request by replacing the placeholders in the template by a value of the attack vector list. Then the connector is invoked with the request.

Connector The Connector is one of two components that communicate with the SUT, with the other being some specific analyzers. It takes the connection settings from the configuration and a request given by the handler. Depending on the configuration, the connector could e.g., start a Transmission Control Protocol (TCP) connection and transfer a HTTP request to the remote end. The response of the SUT is recorded and handed back via the Handler to the Runner module.

Analyze Manager The configuration of the test run tells this module which analyzer programs have to be loaded and managed. For every response it obtains from the Runner, the analyze manager calls all of its analyzers with the response object as argument. The analyzers itself convert the response to a metric and return the value to the manager. All response metrics are weighted and summed up to a single value and returned to the Runner module. This value determines the state of each attack iteration, either *pass* or *fail* for a successful (i.e., no error) or failed test (i.e., an error has occurred).

In general, an analyzer consists of two parts: a data capture module and a metric deduction algorithm. The data capture module extracts the required information from the given response object, or during the test execution (e.g., to measure the response time). The metric deduction algorithm takes the information given by the capture module and infers a value from this information. An example is the response size analyzer: The capture module extracts the content from the response object and the metric deduction measures the size of the content in bytes.

4.3 Integration of the Artificial Intelligence System

In order to extend testing with methods of artificial intelligence, some kind of intermediate step is required. An interface both system can understand. Such an idea of an intermediate model was already published in an earlier collaboration work of the author [73] and is called the *System Behaviour Model*.

In our published work we propose a model that is based on a set of individually measured behaviour metrics of the system and the relationship between them. The first step in obtaining the model is to install sensors around the SUT to monitor the behaviour of the system. This can be done by using different available interfaces of the SUT, and derive measurable metrics. Some examples of interfaces that can be monitored are drawn in figure 4.1, like the operating system, internal and external interfaces as well as an user interface used by the application.

The kind of sensors used often depends on the current test stage of the software development life cycle. During component tests direct access to the SUT is available most of the time which allows white-box monitoring approaches. That is, for example, the possibility to monitor the Central Processing Unit (CPU) load, memory usage of the application and the system, coverage of the application, syscall graphs, etc. During system tests, often no access to the host is available and, therefore, other methods to monitor the SUT are required. In such cases it may be possible to analyze the network traffic, timing behaviour, message sizes, etc. If enough access rights are available during test execution, debuggers can be attached to the SUT to monitor application internals. Often, virtualization is used in system architectures. Virtualization provides additional monitoring possibilities of the encapsulated virtualized system, e.g., extending the Java Virtual Machine allows monitoring access to system components via the Java security manager. If the SUT has a Graphical User Interface (GUI), this is one more example of a possible interface to monitor.

To capture information about the system, we identified approaches from the literature that can be used. Willems *et al.* [86] present several monitoring aspects for malware analysis used in the tool *cwsandbox*[1] which can also be used for monitoring the SUT for security tests. Bossert *et al.* [12] use a monitoring approach for reverse engineering of application protocols. The presented usage of monitoring communication channels can be applied for security tests.

After monitoring the SUT behaviour, the metrics have to be converted in some kind of measurable form. We used aspects from the representational theory of measurement to derive measurable metrics. To process the data, five types of measurement scales have to be considered which are nominal, ordinal, interval, ratio and absolute [44].

A *nominal scale* is an unordered set of categories and the captured behaviour belongs to one of the categories (e.g., male or female). An ordered nominal scale is called *ordinal scale* (e.g., true or false). This scale allows to apply comparisons such as a is greater than b. For nominal and ordinal scales no further arithmetic operations such as addition or subtraction are possible. The *interval scale* is an ordinal scale with consistent intervals between the points on the scale (e.g.,

[1] https://mwanalysis.org/ (last accessed: 29.09.2014)

from $-90°C$ to $+90°C$). Arithmetic operations addition and subtraction are possible for interval scales but not multiplication and division. Using constant intervals for an interval scale will lead to a *ratio scale* (e.g., from 0 to 99 years). For ratio scale all arithmetic operations are valid. This is also the case for the *absolute scale*. The absolute scale is a ratio scale but with the count of the number of occurrences (e.g., the Kelvin temperature scale).

Depending on the values derived using the measurement scales further processing might be required. The reason the allowed arithmetic operations are important is because different further calculations of statistical properties are allowed for different measurement scales. Mean, median and mode are used for central tendency statistics in data sets for a single metric. All three are allowed for interval, ratio and absolute scales. For ordinal scales only mode and median are valid whereas for nominal scale only the calculation of the mode is meaningful.

In our published approach, statistical methods for the measurement of central tendency and variability of the derived set of numeric values are used on the metrics of the system behaviour model. As example, metrics based on the CPU usage of the SUT, an absolute scale and a range measure of variability is used to detect the difference between highest and lowest CPU usage within the execution of one test case. For monitoring a crash of the system, e.g., the SUT does not answer any more, a nominal scale with the categories *system crash* and *no system crash* is used. No further statistical methods are applied.

4.3.1 Representation of the System Behavior Model

To represent the system behaviour model, we combined the individually derived behaviour metrics to one model which can be used during security test execution for the determination of unexpected behaviour. The form used for captured behaviour depends on the interface. Regardless of the captured behaviour of the interface, as a result a numeric representation for the metric is required for the model. This is used for further processing and deriving one final result.

Given one example: analyzing a log file for abnormal behaviour requires a deduction from the textual representation of the observed behaviour to a measurable metric, e.g., counting the occurrence of various critical keywords. Similar processing is required for system load behaviour, e.g., CPU load, which represents the load during the test execution. Representative metrics have to be extracted from the observation of the CPU load, for example, min/max/average of the load during execution of the security test case.

In our presented approach the system behaviour model is defined as a set M of behaviour metric vectors x_i:

$$M := \{x_i | x_i \in \mathbb{R}^n, i = 1, 2, \dots l\} \tag{4.1}$$

where n is the number of behaviour metrics derived from the captured system behaviour for the specific system (i.e., the number of the used analyzers) and l is the number of learning cases used to build the system behaviour model.

During the security test execution, the single metrics are collected after each executed security test case and a measurable form is derived. The measurable form is combined as a vector $t_j \in \mathbb{R}^n$ where j is the index of the security test case. The vector t_j represents the specific system behaviour under security test case j.

The SUT is formally defined as function $f(p_i) = b_i$. The result of the function f is the captured behaviour b_i of the system for the learning case p_i. Further, a function $g(b_i) = x_i$ is required which converts the captured behaviour b_i to a measurable form as vector x_i for the learning case p_i. Testing the software f is defined as $f(q_j) = c_j$ where q_j is the input test data for the SUT. c_j

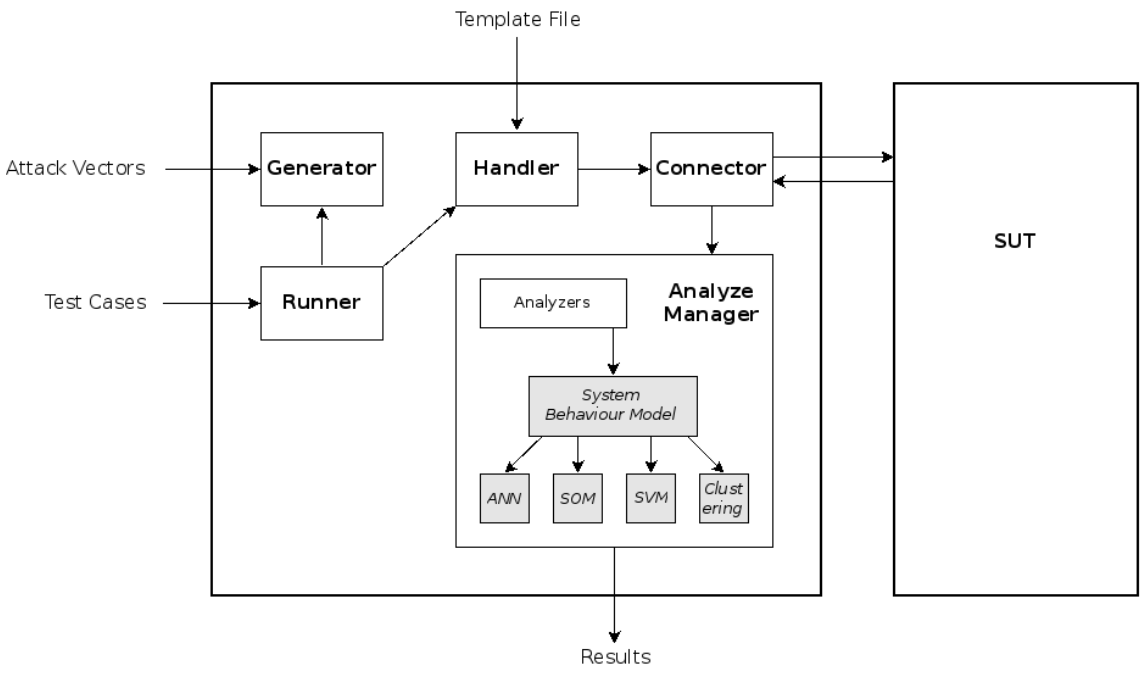


Figure 4.4: Architecture of the extended fuzzolution framework. (based on [74])

is the specific behaviour of the SUT for the test data q_j and the measurable form t_j is determined by $g(c_j) = t_j$.

The system behaviour model and the extraction of behaviour metrics are generic and can be used by various machine learning methods. In the following section, this approach will be used to train the machine learning methods introduced in section 3.

4.3.2 Integration of the Prototype Implementations

The goal of this thesis is to analyze AI methods to improve the detection rate of the fuzzolution tool. The results of a fuzz test will be pre-analyzed by the proposed methods which will reduce the required amount of manual work significantly. The following method is a general idea to improve this situation.

Each fuzzing run is split up into two phases, both similar to a fuzzing run described in the previous section 4.2. During the first phase the generators are requested to exclusively return functional testing data. This is a type of data that is not intended to break the system, like a random integer number when asked for an integer user identification number. The requests based on this valid data, called *learning cases*, are regularly constructed by the handler and sent to the SUT. The second phase remains as the original test run with sending of attack vectors and trying to break the system.

The results of the analyzers are handled differently in both phases. During the first phase, called *learning phase*, the results of the analyzers is used to build the System Behaviour Model and given to a machine learning method. After the learning phase is complete, this model represents the behaviour of the SUT under "normal" working conditions. The results of each *test case* in the second phase are likewise converted to a behaviour vector and given to the machine learning method. The AI method compares the given vector to the previously trained model and determines if the vector is an outlier. If the vector is classified as an outlier this yields some information about possible abnormal behaviour. A vector describing abnormal behaviour can belong to an attack string that probably exploits a security flaw in the SUT.

Figure 4.4 and illustrates the extended fuzzolution architecture including the machine learning methods (see figure 4.2 for the original image). The output of the analyzers inside the Analyze Manager module is used to build the System Behaviour Model. The model is then handed over to the four prototype implementation of the methods: Artificial Neural Network (ANN), Self-Organizing Map (SOM), Support Vector Machine (SVM) and Clustering. They deduce further information about the test cases and return the overall result.

Implemented in the fuzzolution project, this method will improve the overall results of a fuzz test. Instead of manually selecting weights for the analyzers and returning a simple metric, the result would be a metric describing the similarity of each test case compared to the automatically learned, normal system behaviour. This result can be used to select the most ”interesting” cases for further analysis.

5 Setup of Test Environment of the Proof of Concept Implementations

In order to evaluate several machine learning methods and get meaningful results, it is necessary to test the methods against the same benchmarking tool and compare the results. In the following sections the test setup will be explained. A detailed explanation of the prototype implementations for each method and the results will be presented in chapter 6.

5.1 Setup of Test Environment

The actual testing and behaviour model extraction was done with the help of the fuzzing software ”fuzzolution” [1]. It was configured to test all sub applications of WAVSEP and Zed Attack Proxy - Web Application Vulnerability Examples (ZAPWAVE).

The analyzers where chosen following the suggestions of originating paper [73] to reflect a wide range of metrics. This allows the deduction of a very detailed system behaviour model. In particular, the analyzers used in the test setup were: [73]

- **Response time analyzer:** Measuring the response time that was needed for the request to be handled by the SUT. This means increased processing time from the simulated attack request can be detected.
- **Response size analyzer:** This analyzer uses the size of the response which can yield some information about unexpected responses if the returned response differs from the normal behaviour.
- **Response message analyzer:** The response message analyzer uses the text within the response for further processing. It checks the content of the response for suspicious strings, e.g., error messages. To make decisions about captured text behaviour, a metric has to be derived. A simple approach of counting the occurrence of predefined keywords in the response was used. Example keywords used for the security test execution are: `exception`, `bad`, `missing`, `fatal`, or `segmentation fault`.
- **Response classification analyzer:** The response classification analyzer is based on methods of the field of information retrieval. It uses text indexing to build an *index* of the response data during the learning phase. In the testing phase it starts a *query* on the index for every test case by comparing the response data to the existing data. By deducing a similarity metric from this comparison, it can tell whether the current response is similar to a learned and known response or not. The advantage of this method is that the metric is deduced from the content of the response. Two responses with the same length can get assigned a different value from this analyzer.

[1] http://security.inso.tuwien.ac.at/esse-projects/fuzzolution/ (last accessed: 29.09.2014)

- **Crash analyzer:** The crash analyzer checks if a crash of the SUT occurred by trying to establish a connection to the SUT. The analyzer performs a full TCP handshake using the TCP ports of the SUT. It returns a boolean value, thus if the handshake is successful, the system is regarded as “up”, otherwise the system is considered to have “crashed”.
- **CPU load analyzer:** This analyzer monitors the CPU load of the processes in the SUT to detect heavy load during or after executing a test which could indicate a Denial of Service (DoS) attack.
- **Memory analyzer:** This analyzer monitors the size of the allocated memory of the processes in the SUT to detect changes during or after executing a test case.

The deduced values for each iteration are then stored as mathematical vector for further analysis. All vectors gathered during one test run combined form the *System Behaviour Model*. As example, one resulting training vector during the test of the WAVSEP tool was:

$$x_{20} = (6.00, 356.00, 0.40, 43.57, 1.00, 0.00, 0.00) \tag{5.1}$$

In comparison to this, one test vector of the same test run:

$$t_{48} = (12889.05, 356.00, 0.40, 43.57, 1.00, 0.15, 0.00) \tag{5.2}$$

This vector is basically an encoded form of all the metrics that the analyzers returned after the examination of the response of the SUT. They can be decoded again to the following values for each analyzer:

Analyzer	x_{20}	t_{48}	remark
Response time analyzer	6.00	12889.05	ms
Response size analyzer	356.00	356.00	Byte
Response message analyzer	0.40	0.40	
Response classification analyzer	43.57	43.57	
Crash analyzer	1.00	1.00	no crash
CPU load analyzer	0.00	0.15	load
Memory analyzer	0.00	0.00	% usage

Table 5.1: Detailed values for each analyzer during one example test case.

Finally, the behaviour data is used to train one of the following machine learning methods. The requested task of these methods is to analyze the training vectors and, based on the gathered knowledge from the training phase, detect outliers in the attack vectors. These outliers represent behaviour that was not encountered during the functional testing. Therefore it is considered interesting since such behaviour could lead to possible vulnerabilities in the tested software.

The ability of a machine learning method to detect samples in the test set that are significantly different from those in the training set is sometimes referred to as *novelty detection* [6].

5.2 Description of System Under Test

As system under test, two different benchmarking tools were selected: the ”WAVSEP” [2] as well as the ”ZAPWAVE” [3]. Both tools are designed for evaluating the coverage and accuracy of security testing tools. They are written as Java servlets and can therefore easily be deployed onto any servlet engine, e.g., Apache Tomcat [4]. Since both platforms are based on web pages, they can easily be accessed by any HTTP client. This has several advantages. First of all, the HTT-Protocol is a very common form of data transmission, thus many implementations and frameworks already exist. Many security scanning tools already understand this protocol, so does the fuzzolution project. Second it is a text-based protocol, which gives a big advantage when it comes to automated scanning. The pages of the applications of both tools can easily be parsed, so the configuration of the testing tool can be done without much manual work.

The tools offer a number of different small applications, against which the actual testing is done. These programs may contain flaws that lead to vulnerabilities and their output can be anything between nothing and, for example, a full exception including a stack trace. When evaluating a security testing tool, the goal is to detect all vulnerabilities in each application, if there exists any and do not report anything if the application contains none.

In his blog called *sectooladdict*, Shay Chen publishes a score chart of vulnerability scanners in yearly intervals starting in 2010. Chen is the CTO of Hacktics ASC and made a name as prominent blogger, security researcher and experienced speaker [21]. The chart is called *The Web Application Vulnerability Scanners Benchmark* [5]. In this blog, Chen compares the qualities accuracy, coverage, versatility, adaptability, feature and price of several commercial and open source vulnerability scanners. Taken from the introduction section of the latest benchmark report, the project seems to be getting a lot of attention from organizations in the financial and technology sector using it as reference for their decisions.

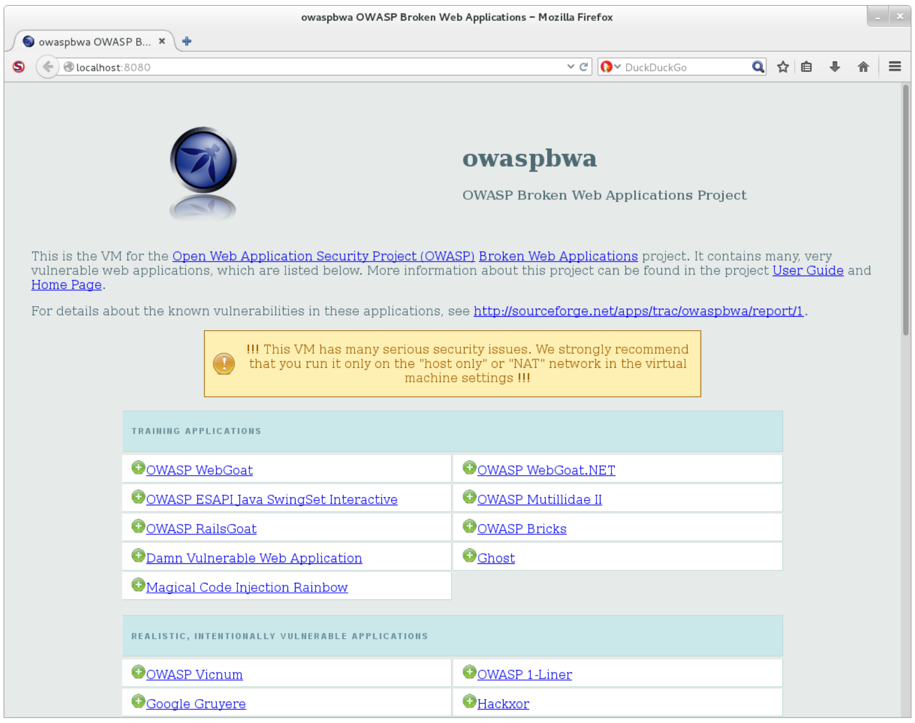
In the last edition from February 2014, a total of 63 tools were compared in a very detailed and thorough fashion. All scanners were tested against the benchmarking platform WAVSEP and included test cases from the ZAPWAVE project. The overall leader in this years comparison is a software of the HP Application Security Center, called *WebInspect*.

To evaluate the prototype implementations the same test environment was used as in the scanner benchmark report, consiting of the platforms WAVSEP and ZAPWAVE.

The OWASP team maintains a project called the ”Broken Web Applications Project”. This is a preconfigured virtual machine image including WAVSEP and ZAPWAVE, among others. With the start of the virtual machine an Tomcat instance is started automatically. On this servlet engine both tools are already deployed. After the configuration of the port forwarding to the local machine, the tools can be accessed e.g., via the URL `http://localhost:8080` and are ready for testing. Figure 5.1 shows the entrance page of the Broken Web Applications including an overview of all available tools in this virtual machine.

There is one big advantage of having a physically or virtually separated and dedicated SUT environment. Presuming, the host machine has enough hardware resources to support both with ease, there will be hardly any influences between the system running the testing software and the machine running the SUT. This is of importance during test execution and measuring of system

[2] https://code.google.com/p/wavsep/ (last accessed: 29.09.2014)
[3] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project (last accessed: 29.09.2014)
[4] https://tomcat.apache.org/ (last accessed: 29.09.2014)
[5] http://sectooladdict.blogspot.com (last accessed: 29.09.2014)



owaspbwa

OWASP Broken Web Applications Project

This is the VM for the Open Web Application Security Project (OWASP) Broken Web Applications project. It contains many, very vulnerable web applications, which are listed below. More information about this project can be found in the project User Guide and Home Page.

For details about the known vulnerabilities in these applications, see http://sourceforge.net/apps/trac/owaspbwa/report/1.

!!! This VM has many serious security issues. We strongly recommend that you run it only on the "host only" or "NAT" network in the virtual machine settings !!!

TRAINING APPLICATIONS

OWASP WebGoat	OWASP WebGoat.NET
OWASP ESAPI Java SwingSet Interactive	OWASP Mutillidae II
OWASP RailsGoat	OWASP Bricks
Damn Vulnerable Web Application	Ghost
Magical Code Injection Rainbow	

REALISTIC, INTENTIONALLY VULNERABLE APPLICATIONS

OWASP Vicnum	OWASP 1-Liner
Google Gruyere	Hackxor

Figure 5.1: The OWASP Broken Web Applications Project page.

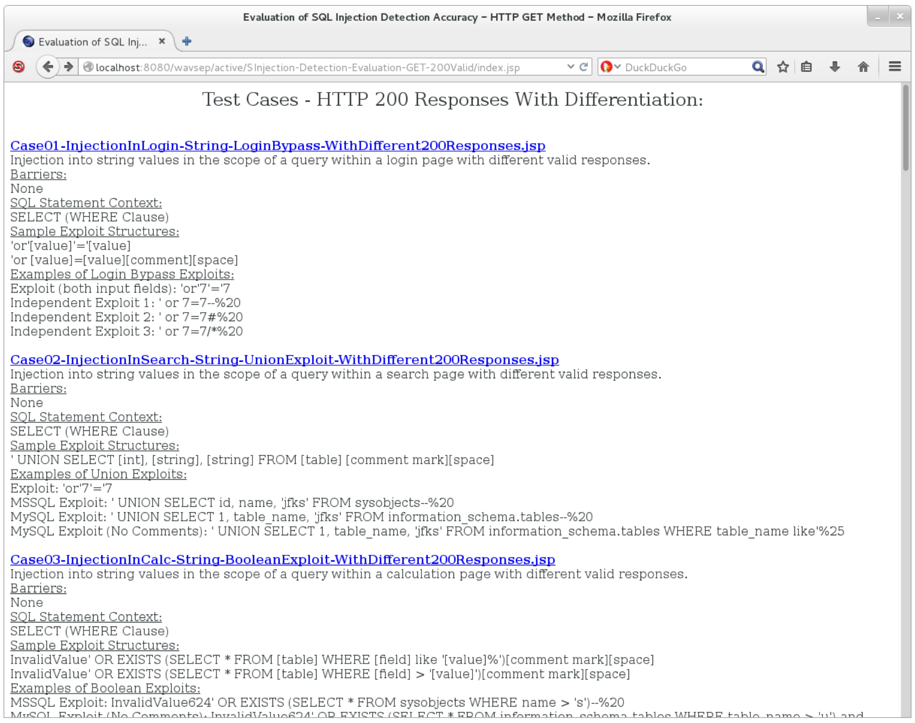
parameters. As example, a high CPU-load in the host machine caused by the testing software will not be visible inside a guest virtual machine. On the other hand, a high CPU-load inside the guest will most likely be caused by the software of the SUT. This circumstances help to reduce the external influences, and thereby lead to a higher significance of the evaluation results.

5.2.1 WAVSEP

The Web Application Vulnerability Scanner Evaluation Project contains a collection of vulnerable web pages categorized into different security flaws. These known flaws can be used to evaluate the effectiveness of security testing tools. Among others, these vulnerabilities are: Path Traversal, Remote File Inclusion, XSS and SQL Injection. For the evaluation of the methods in the thesis, the category SQL Injection was selected, since these kind of vulnerabilities are a major risk to web applications (as discussed in section 2.1) and since injection attacks are targeted directly on the system, they are well suited for an automated testing approach. Moreover, a successful exploit can typically be detected directly, either through the output of the test subject, through timing behaviour, or some similar mechanics.

The SQL Injection web page of the WAVSEP platform is divided into four parts, each with a different responses. These are:

- **Erroneous 500 Responses**: These pages return a HTTP 500 response code including an error message, if they cannot complete the task.
- **Erroneous 200 Responses**: These pages return a standard HTTP 200 response code including an error message, if they cannot complete the task.
- **200 Responses With Differentiation**: These pages return a standard HTTP 200 response code including some valid message, depending on the result of the execution.


Evaluation of SQL Injection Detection Accuracy – HTTP GET Method – Mozilla Firefox
Evaluation of SQL Inj...
localhost:8080/wavsep/active/SInjection-Detection-Evaluation-GET-200Valid/index.jsp
DuckDuckGo

```
Test Cases - HTTP 200 Responses With Differentiation:

Case01-InjectionInLogin-String-LoginBypass-WithDifferent200Responses.jsp
Injection into string values in the scope of a query within a login page with different valid responses.
Barriers:
None
SQL Statement Context:
SELECT (WHERE Clause)
Sample Exploit Structures:
'or'[value]'='[value]
'or [value]=[value][comment][space]
Examples of Login Bypass Exploits:
Exploit (both input fields): 'or'7'='7
Independent Exploit 1: ' or 7=7--%20
Independent Exploit 2: ' or 7=7#%20
Independent Exploit 3: ' or 7=7/*%20

Case02-InjectionInSearch-String-UnionExploit-WithDifferent200Responses.jsp
Injection into string values in the scope of a query within a search page with different valid responses.
Barriers:
None
SQL Statement Context:
SELECT (WHERE Clause)
Sample Exploit Structures:
' UNION SELECT [int], [string], [string] FROM [table] [comment mark][space]
Examples of Union Exploits:
Exploit: 'or'7'='7
MSSQL Exploit: ' UNION SELECT id, name, 'jfks' FROM sysobjects--%20
MySQL Exploit: ' UNION SELECT 1, table_name, 'jfks' FROM information_schema.tables--%20
MySQL Exploit (No Comments): ' UNION SELECT 1, table_name, 'jfks' FROM information_schema.tables WHERE table_name like'%25

Case03-InjectionInCalc-String-BooleanExploit-WithDifferent200Responses.jsp
Injection into string values in the scope of a query within a calculation page with different valid responses.
Barriers:
None
SQL Statement Context:
SELECT (WHERE Clause)
Sample Exploit Structures:
InvalidValue' OR EXISTS (SELECT * FROM [table] WHERE [field] like '[value]%')[comment mark][space]
InvalidValue' OR EXISTS (SELECT * FROM [table] WHERE [field] > '[value]')[comment mark][space]
Examples of Boolean Exploits:
MSSQL Exploit: InvalidValue624' OR EXISTS (SELECT * FROM sysobjects WHERE name > 's')--%20
MySQL Exploit (No Comments): InvalidValue624' OR EXISTS (SELECT * FROM information_schema.tables WHERE table_name > 'u') and
```

Figure 5.2: The WAVSEP web interface displaying some application descriptions.

- **Identical 200 Responses**: These pages return a standard HTTP 200 response code including a fixed valid message. Thus, successful exploits can only be detected by blind SQL Injections.
- **False Positive Injection**: These pages return a HTTP 500 or 200 response code, along with an erroneous message no matter the input. This tests can be used to challenge scanners with weak exploit detection mechanisms.

All these application are called via an URL and their arguments can be set either by GET or POST method. An example URL with GET parameters from third application of the category *200 Responses With Differentiation* is :

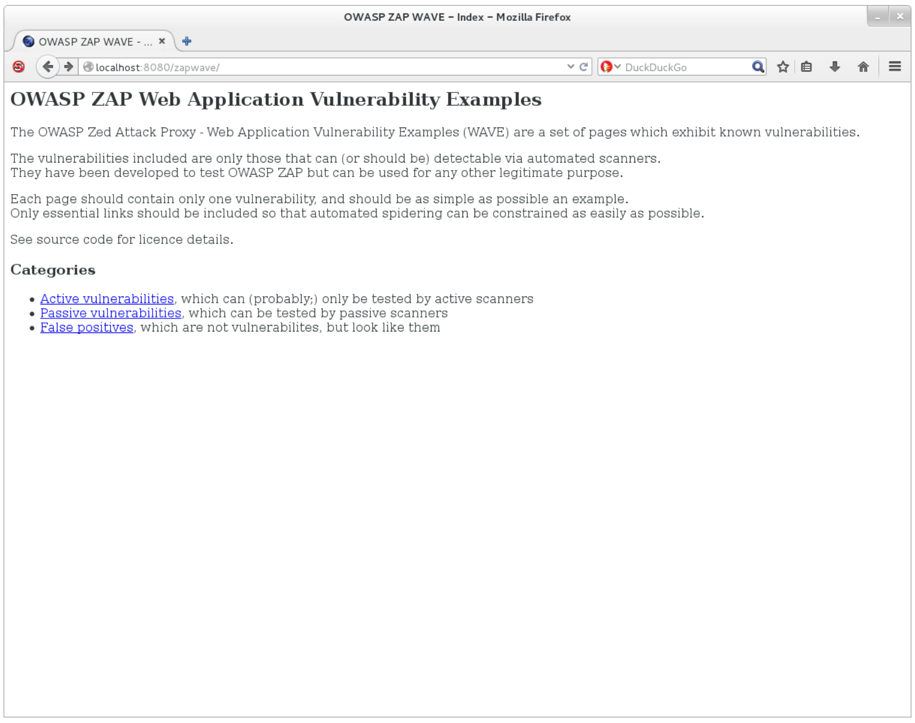
```
http://localhost:8080/wavsep/active/SInjection-
   Detection-Evaluation-GET-200Valid/Case03-
   InjectionInCalc-String-BooleanExploit-
   WithDifferent200Responses.jsp?username=textvalue
```

Listing 5.1: An example attack string for an application of the WAVSEP platform.

Listing 4.2 is the associated template for this attack string. The description page of this application along with the link to its URL can be seen in the screenshot displayed in figure 5.2.

5.2.2 ZAPWAVE

The Zed Attack Proxy - Web Application Vulnerability Examples test platform is a spin off of the OWASP Zed Attack Proxy (ZAP) project. This attack proxy is a similar to the fuzzolution framework. Both are security testing tools and can be used for finding vulnerabilities in applications.



OWASP ZAP Web Application Vulnerability Examples

The OWASP Zed Attack Proxy - Web Application Vulnerability Examples (WAVE) are a set of pages which exhibit known vulnerabilities.

The vulnerabilities included are only those that can (or should be) detectable via automated scanners.
They have been developed to test OWASP ZAP but can be used for any other legitimate purpose.

Each page should contain only one vulnerability, and should be as simple as possible an example.
Only essential links should be included so that automated spidering can be constrained as easily as possible.

See source code for licence details.

Categories

- Active vulnerabilities, which can (probably;) only be tested by active scanners
- Passive vulnerabilities, which can be tested by passive scanners
- False positives, which are not vulnerabilites, but look like them

Figure 5.3: The ZAPWAVE web interface displaying the application categories.

However, the ZAP is designed for web applications only, where the fuzzolution tool can be used in any environment, given an appropriate configuration.

The ZAPWAVE platform is a small set of web pages with known vulnerabilities that can be detected via automated scanners. These pages have been developed alongside the ZAP to have a set of pages that can be used for testing the proxy.

Available for testing are several applications with different security flaws with the main categories active and passive vulnerabilities. The difference between those two is that an active vulnerability can only be detected by direct testing, i.e., sending requests to the program. Implemented examples are XSS, SQL injections and redirection attacks. Passive vulnerabilities can only be detected indirectly by listening to the traffic between a client and the vulnerable server. The examples contained in the ZAPWAVE tool are information leakage and session handling defects.

Similar to the WAVSEP environment, for each application exists a variant for GET and for POST parameter handling. An example application URL that contains an SQL Injection can be accessed via:

```
http://localhost:8080/zapwave/active/inject/inject-
    sql-url-basic.jsp?name=textvalue
```

Listing 5.2: An example attack string for an application of the ZAPWAVE platform.

A screenshot of the starting page of the ZAPWAVE platform is shown in figure 5.3. Note that applications are linked that should return false positive results. However in the version used for the evaluation, those were not implemented yet.

6 Evaluation of Proof of Concept Implementations

This chapter presents the details of the implementations and the results of the evaluations of all methods mentioned in chapter 3. Each method has its design explained, followed by a detailed description of the parameters of the implementation. The evaluation consists of an analysis of the runtime of each test run, as well as the overall results, particularly the correctly identified test cases.

6.1 Evaluation Method

To evaluate the output of the methods, a manual analysis was done on all results of the test runs. The manually detected outliers were compared to the marked vectors of the methods. Four cases can occur by this comparison:

Manually marked, automatically marked: This case counts as *correct* case. The machine learning method correctly detected an outlier, according to manual analysis.

Manually marked, automatically not marked: The method failed to detect an existing outlier, leading to a *false negative* case.

Manually not marked, automatically marked: The method detected an outlier but was wrong in its decision. A *false positive* case.

Manually not marked, automatically not marked: Another *correct* case. The algorithm did not detect any outlier and none was present.

Both cases of correct detection summarized and the number of false positive as well as the number of false positive cases give a triple that can be used for comparison. All three values can be summarized to the total number of vectors.

The test data of both tools, WAVSEP and ZAPWAVE, consisted of 76 complete test runs with a total of 21315 vectors.

6.2 Artificial Neural Network (ANN)

The implementation of the Artificial Neural Network prototype follows the explanations of section 3.2. The design decisions are based on the proposed design of Augusteijn *et al.* in their publication ”Neural network classification and novelty detection” [6].

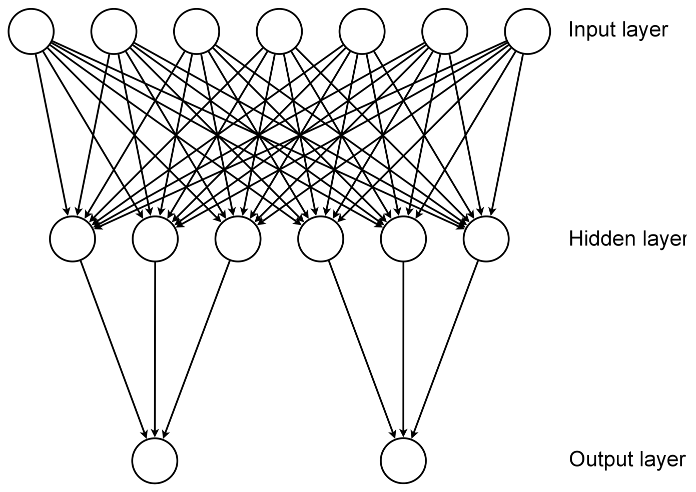


Figure 6.1: Artificial Neural Network design of the prototype implementation.

6.2.1 Design Decisions for the Implementation

The implemented network design is based on the network structure used by Augusteijn [6] and is schematically illustrated in figure 6.1. The number of input neurons correspond to the dimension of the behaviour vectors, such that every input neuron processes one vector. The prototype implementation therefore consists of seven input neurons. The input neurons are fully connected to the first layer of hidden neurons. In total there are three layers of hidden neurons, each fully connected to the next layer. In the implementation every hidden layer has six neurons. The output layer of the network consists of two neurons. The third and last layer of hidden neurons is connected to the output layer as follows: The first half of the hidden neurons is connected to the first output neuron, whereas the second half of the hidden neurons is connected to the second output neuron.

Following the suggestions of Augusteijn [6], as activation function the *hyperbolic tangent function*

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{6.1}$$

was chosen, as adder function the *weighted sum function* (see formula 3.1). The learning rule was standard *back-propagation*.

The software prototype was implemented in Java using the framework ”Neuroph” [1]. It offers all functions for building, training and testing an artificial neural network.

For every test run, the neural network was trained with the vectors representing the normal behaviour of the SUT ($F = \{f_1, \dots, f_k\}, f_i \in \mathbb{R}^n$) with a required output of $(1, -1)$. The network was trained using the parameter γ. The training continued until either 20,000 iterations were done or until the *mean-squared error* of the network was 0.01 or below.

After the network was successfully trained, each training vector $f_i \in F$ was applied to the network and the euclidean distance to the vector $(1, -1)$ was calculated. Let $F' = \{f'_1, \dots, f'_k\}, f'_i \in \mathbb{R}$ be the set of training vector differences. The extreme values of these distances $min(F')$ and $max(F')$ were stored for later use. Then the testing vectors $T = \{t_1, \dots, t_l\}, t_i \in \mathbb{R}^n$ were applied to the network and again the euclidean distances of their output vectors to the coordinates $(1, -1)$ was calculated and stored as set $T' = \{t'_1, \dots, t'_l\}, t'_i \in \mathbb{R}$.

[1] http://neuroph.sourceforge.net/ (last accessed: 29.09.2014)

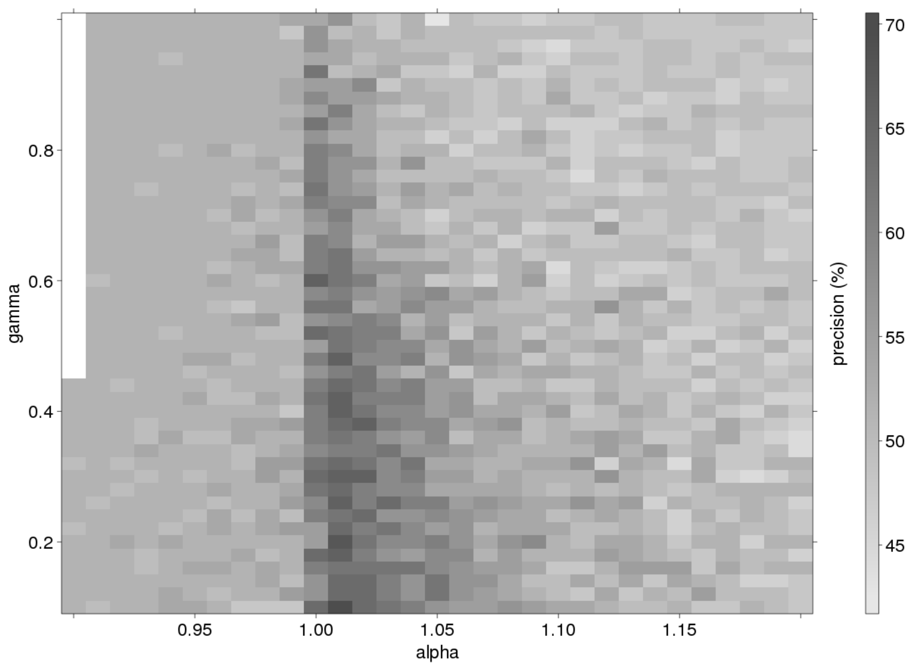


Figure 6.2: Artificial Neural Network parameter evaluation.

The outlier detection included the parameter α and was defined such that for every t' the following check was made:

$$outlier(t_i) = \begin{cases} 1 & \text{if } t'_i > \alpha \; max(F') \\ 1 & \text{if } t'_i < \frac{1}{\alpha} \; min(F') \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

As described above, the neural network was trained using the learning rate $\gamma \in [0, 1]$. This parameter influences the rate at which shifts in the synaptic weights are performed. The lower the value, the more accurate the trainings but the process requires more time to finish and possibly even diverges. Choosing a greater value may lead to a faster learning process convergence but will affect the accuracy of the network.

The second parameter α is used by the outlier detection. Based on the results of the training phase it defines the α-environment of outliers. If a vector belongs to this environment it is classified as outlier. As simple example, suppose the distances of the training vectors range from 2 to 7, then an $\alpha = 1.1$ defines the outlier environment of $\{x \mid (x < 1.81) \vee (x > 7.7)\}$.

To find the optimal parameters for the designed artificial neural network, several combinations of both parameters were tested. Values for the learning rate γ where taken from the range of 0.10 to 1.00 in steps of 0.02, i.e., :

$$A = \{0.10, 0.12, 0.14, \ldots, 0.98, 1.00\}$$

The range for the outlier parameter α used in the test cases was defined to be $[0.9, 1.2]$, since preliminary tests indicated that no useful further results could be extracted outside this range. Thus the values for α were taken from its limits 0.90 to 1.20 in steps of 0.01, i.e., :

$$B = \{0.90, 0.91, 0.92, \ldots, 1.19, 1.20\}$$

The test subject of the parameter tests is described in section 5.2 and was used for all tests. The combination of both parameter sets $A \times B$ give a total of 1426 test cases to evaluate the neural network. The output of each test case is the percentage p of the *correctly* identified test cases (see section 5.1 for details on the evaluation of test runs). The result is a three dimensional data set where each entry is a triple $v = (\alpha, \gamma, p)$.

The data set visualised in figure 6.2. The x-axis displays the range of the outlier parameter α where on the y-axis is the range of the learning rate γ and the z-axis is the resulting precision p of each test case at (α, γ). A representation of the third dimension is given by the levels of gray on each coordinate and is described by the color bar on the right.

As the figure clearly states, an α-value lower than 1.00 does not yield satisfying results. A precision rate of about 50% can even be scored by a simpler guessing algorithm. Starting from 1.00 to about 1.03 is an high density area of greater precision values, slowly fading with an increasing γ-value.

Following this figure it is safe to assume that good estimates of the parameters for α are about 1.01 while γ should not be higher than 0.4.

6.2.2 Results of the Evaluation

Using the gained information about the parameter values for the outlier detection based on neural networks a final test run can now be done. The parameters for the test run where chosen as $\alpha = 1.01$ and $\gamma = 0.1$. Since the training of the neural network is a non-deterministic algorithm, it is required to complete the test run severals times and extract statistical information from these runs.

The performance of this method on all test cases can be seen in table 6.1:

	mean		median		std. dev.	
correct	13805.00	64.76%	13767.50	64.59%	582.03	2.73%
false positive	3684.33	17.28%	3655.00	17.14%	382.42	1.79%
false negative	3825.67	17.94%	3864.50	18.13%	478.73	2.25%

Table 6.1: Evaluation result of the Artificial Neural Network method.

6.3 Self-Organizing Map (SOM)

The self-organizing map is based on the introduction section 3.3. The design of the implementation is based on an earlier work of the author [73].

6.3.1 Design Decisions for the Implementation

The software prototype was implemented in Java using the ”Java SOMToolbox” [2]. This toolbox is developed at the Institute of Software Technology and Interactive System at the Vienna University of Technology. It offers tools for training a SOM and its framework was used for the implementation of the novelty detection.

[2] http://www.ifs.tuwien.ac.at/dm/somtoolbox/ (last accessed: 29.09.2014)

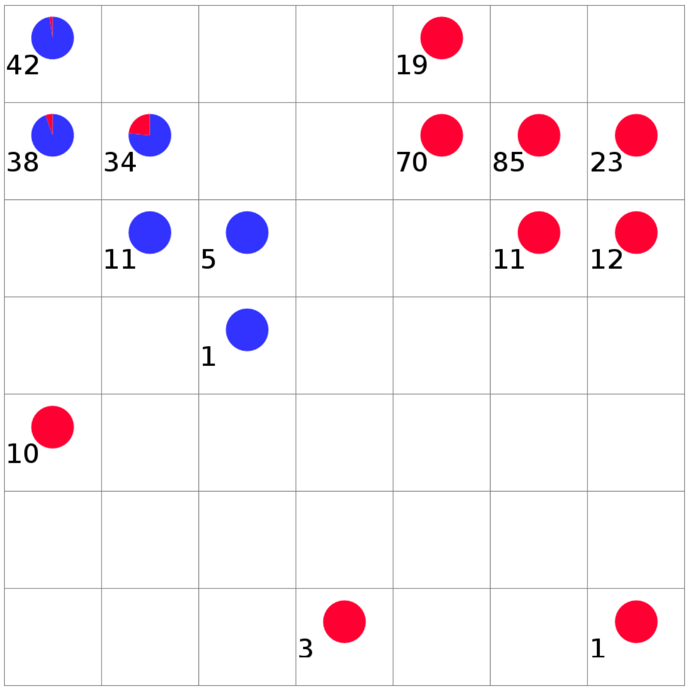


Figure 6.3: A visualisation of a test run evaluated with a SOM.

Both training and testing vectors of one test run are used in the training of the SOM in order to create a map that spans over all input vectors. After the training was completed all units and their assigned vectors were inspected in detail. A unit that contained training vectors was considered as normal behaviour of the SUT. However, if a unit consisted solely of testing vectors, it indicated that the vector of the unit itself was different in a way. Since such a unit did not contain any training vectors, all vectors belonging to such a unit were marked as outliers.

An example of a SOM is given in figure 6.3. The picture shows the visualisation of one SOM that was created during the evaluation. It has the size of 7×7 units and contains a total of 355 vectors. The pie charts inside the units represent the distribution of training and test vectors belonging to this unit. The blue parts represent training vectors, the red parts of the pie represent test vectors. All units with plain red dots are considered interesting. Additionally, the amount of empty units between two filled ones *can* be a clue for the distance of the unit vectors from each other, but this is not generally valid. For a precise visualisation of the distances between the units, an additional distance indicator would be needed.

The algorithm behind the Self-Organizing Map is controlled by several parameters. The structure of the map is mainly defined its size and the number of neighbours for each unit. The number of neighbours was decided to be the common four neighbours, leading to square shaped units in the map. Its width and height is defined by the two parameters x and y respectively. The selection of the size of the map has an influence on the overall performance of the outlier detection. If the map is too small, it is not possible for the units to spread out as needed, leading to a poor representation of the input data. This case may lead to high number of false negatives, since it is highly possible that at least one training vector is contained in all units. If the map is too large, the training vectors will be distributed very sparse in a large area, leading to a possible falsification of the outlier detection. This could lead to many false positive classifications, since there is possibly not enough training data available that helps classifying the test vectors.

In training the map, two other values are of importance. The first is the number of iterations with which the map is trained. The second is the learning rate α, which was defined by formula of the

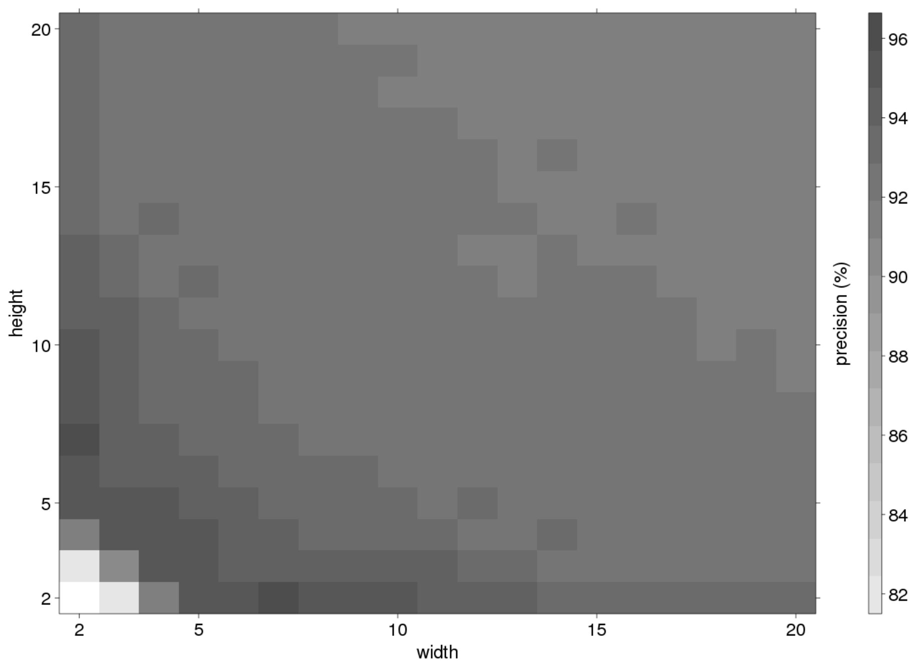


Figure 6.4: Self-Organizing Map parameter evaluation.

neighbourhood function (see formula 3.14). Both influence the unfolding of the map and have to be chosen carefully. If the neighbourhood function converges too fast or the number of iterations is chosen too small, the units in the map may not had time to expand and thus can not represent their actual distances. The documentation of the *somtoolbox* suggest a default value of $\alpha = 0.75$ and the number of iterations to be about 5 times the number of input vectors. Since the test setup of section 5.1 was used in testing the performance of the SOM method, the number of iterations was fixed prior to testing. Each test case uses about 500 vectors and to be on the safe side, the number of iterations was determined to be $10,000$. With such a high number another advantage was gained: Preliminary tests indicated that the learning rate α had virtually no influence on the overall performance with such a high number of iterations, since the map hat enough time to unfold.

The remaining two variables x and y for the width and height of the map can theoretically be chosen from the domain of natural numbers. However a SOM with only one unit is not applicable and the same holds for too large maps. A selection of the set $A = \{2, 3, \ldots, 20\}$ for both parameters have proven useful during earlier tests. Testing each x with each y leads to $|A \times A| = 361$ test cases for the parameter evaluation. As before, the output of each test case is the percentage p of correctly identified vectors. The result is a three dimensional data set where each entry is a triple $v = (x, y, p)$.

The results of the evaluation are illustrated in figure 6.4. The x-axis of the diagram represents the width x of the map, where the height y is represented by the y-axis. The z-axis is the resulting precision p of each SOM with size (x, y). The third dimension is represented by the levels of gray on each coordinate. An overview of the levels gives the color bar on the right side of the diagram.

Clearly visible is a gradient from the upper right corner to the lower left. The precision levels are symmetric along the diagonal (e.g., $(5, 12)$ has the same precision as $(12, 5)$), which supports the property of the SOM that the orientation of the map has no influence on its development. The gradient increases slowly from top right until about a line from $(2, 5)$ to $(5, 2)$, then the precision drops rapidly. The area with the highest precision is curve-shaped and reaches from about $(2, 10)$

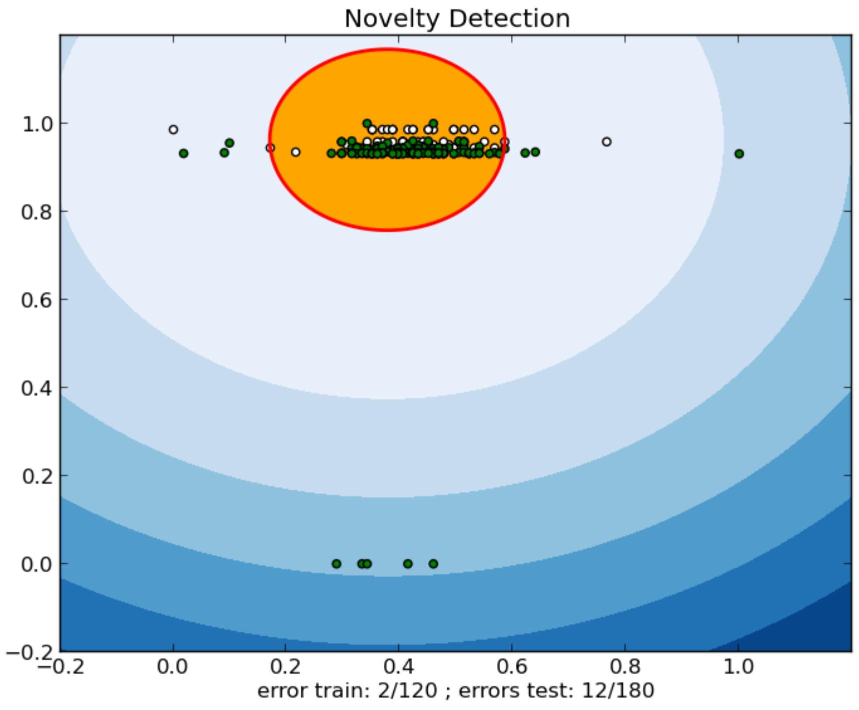


Figure 6.5: An example for an SVM used for novelty detection.

down to $(4, 4)$ with a peak precision at $(2, 7)$. This observations lead to the conclusion, that the size of the map should be somewhere between 10 and 25 units and have a non-square shape.

6.3.2 Results of the Evaluation

Based on the results of the parameter validation above, a final test run was executed. The parameters of the SOM were determined to be $x = 2$ and $y = 7$, which leads to a map with 14 units. Due to the random selection of the vectors for training the map, the SOM algorithm is non-deterministic. However, because of the large number of training iterations, in every case the unit vectors unfolded into the same map and therefore the results are exactly the same.

The performance of this method on all test cases was as follows:

	sum	
correct	20402	95.71%
false positive	745	3.49%
false negative	168	0.78%

Table 6.2: Evaluation result of the Self-Organizing Map method.

6.4 Support Vector Machine (SVM)

The prototype follows the description for Support Vector Machines that has been given in section 3.4, with small adjustments.

6.4.1 Design Decisions for the Implementation

In a traditional training set for SVMs multiple classes are present, but in the case of the unsupervised novelty detection no class labels are available. Lukashevich *et al.* [51] suggested the use of a *one-class SVM* in this case. For their experiments, they used a common type of kernel, a RBF:

$$K(x_i, x_j) = exp(-\gamma \|x_i - x_j\|^2) \tag{6.3}$$

The software prototype was implemented in Python using the module "sklearn" from the *scikit-learn* machine learning project[3]. This module offers a wide range of simple and efficient tools for data mining and analysis.

The "sklearn" Python module supports the use of an parameter μ that controls the size of the area inside the support vectors by setting an upper bound on the fraction of training errors and a lower bound on the fraction of support vectors. A training error occurs if a training vector, which is by definition no outlier, is not supported by any vectors. The value for μ is selected from the interval $(0, 1]$. A second parameter is γ, which defines the kernel coefficient. The *scikit-learn* documentation suggest a default value of $\gamma = \frac{1}{num\_features}$, where $num\_features$ is the dimension of the input vectors. In case of the test setup described in section 5.1 the default value is $\gamma = \frac{1}{7} = 0.143$.

In training of the SVM, an iterative process was selected. Each iteration started with the selection of a value for μ. The training of the SVM with all training vectors formed one test run. A *training error ratio* ε was calculated as the ratio of vectors covered by the resulting support vectors and those not covered. If the error value was below a certain bound (e.g., 0.01) the SVM was accepted and the training terminated. Otherwise the area covered by the support vectors was increased slightly (by decreasing the value of μ) and a new iteration started. Since the initial value of ε was chosen to be very small, it could happen that no SVM was accepted. In this case the value for the training error was increased slightly and the training process started anew. This method assured to have a final result with the least possible ε.

After a SVM has been accepted, the determination of outlying test vectors is a trivial process: If a test vector is not covered by any support vector it must be different from the training data and is therefore considered interesting and marked by the method.

Figure 6.5 illustrates a simplified test run. The dimension of the training and test vectors have been reduced to two, such that they can be drawn on the plane. Clearly visible is the kernel function as red circle and the covered area in orange. Training vectors are drawn as white dots and test vectors are drawn as green dots on the map. It can be seen that the kernel function has been trained with the training vectors, as almost all white dots are inside the circle. A small training error ratio of $2 : 120$ is accepted. Test vectors that can be considered interesting are those twelve green dots outside the kernel function.

Due to the design of the method described above, many parameters of the algorithm are determined automatically. The training error ratio ε started initially with a value of 0.01. The second parameter

[3] http://scikit-learn.org/ (last accessed: 29.09.2014)

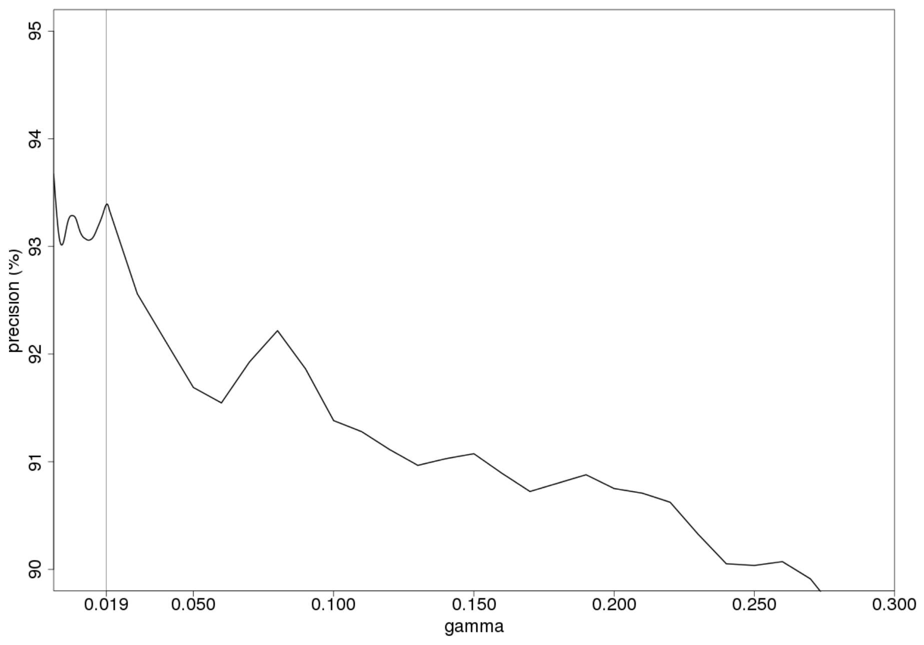


Figure 6.6: Support Vector Machine parameter evaluation.

was initialised with the value of $\mu = 0.3$ and was decreased by 0.001 if the error ratio was not satisfied by the resulting SVM. If the value of μ at some point reached 0, then the value for ε was increased by 0.01. Then μ was reset to 0.3 and the process started over until a SVM with a low enough error rate was found.

The third parameter γ of the SVM algorithm is thus not fixed and appropriate values have to be determined. As mentioned above, a suggested value for γ is the inverse of the dimension of the test vectors. With this reference value and some preliminary testing, this leads to a suggested interval of about $[0.0002, 0.3]$. The selected values for the parameter testing where split in two sets with different steps. From 0.0002 to 0.2 each step increased by 0.0002. From 0.02 to 0.3 the parameter was incremented by 0.01. This lead to a combined set of:

$$A = \{0.0002, 0.0004, \dots, 0.0198, 0.02, 0.03, \dots, 0.29, 0.3\}$$

As before, the test environment described in section 5.1 was used in determining optimal values for γ. The number of test cases was given by the size of the set, therefore 128 tests were executed. The output of each case is again the percentage p of the correctly identified outliers. Since there was only one parameter to evaluate, each resulting vector is a tuple $v = (\gamma, p)$.

The set of resulting vectors is displayed in figure 6.6. In general can be said that there is a slight decrease of precision with increasing γ values. Viewed in detail, the resulting precision stays about constant high in the interval from $(0, 0.019]$ and decreases afterwards. However the decrease is not very significant since the precision of the SVM drops from about 93% to about 91%. Nevertheless this leads to the conclusion that a γ of less than 0.02 is a good estimate value.

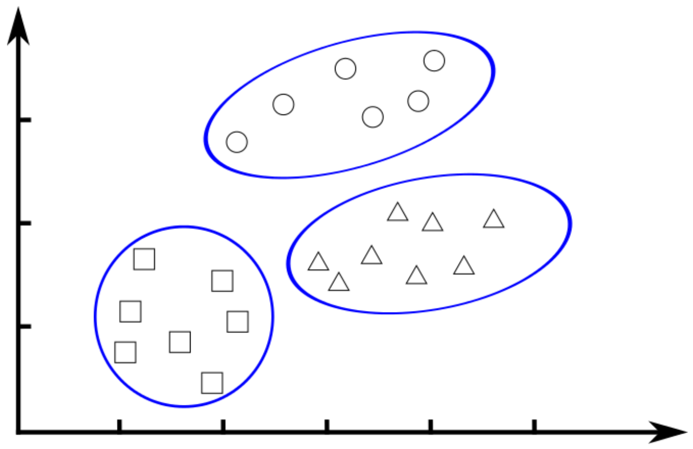


Figure 6.7: An example for an c-means clustering used for outlier detection.

6.4.2 Results of the Evaluation

With the results of the parameter evaluation, the final test run for the outlier detection based on the Support Vector Machine can be executed. The parameter γ for the test run was chosen to be 0.019. Due to the fact that the SVM is a deterministic algorithm, one test run suffices for the final results, which are:

	sum	
correct	19969	93.68%
false positive	1236	5.79%
false negative	110	0.51%

Table 6.3: Evaluation result of the Support Vector Machine method.

6.5 Clustering Algorithm

The implemented clustering algorithm prototype follows the description of the Hard c-Means clustering in section 3.5. It is further based on the publication ”Discovering cluster-based local outliers” as described by He *et al.* [28].

6.5.1 Design Decisions for the Implementation

In the prototype both training and test data sets are used for clustering. A crucial factor of the method is the resulting numbers of clusters c. The determination of the factor c is done with respect to the size of the input data. In the implemented approach the number of clusters were defined as:

$$c = \left\lfloor \frac{1}{\alpha} |M| \right\rfloor \tag{6.4}$$

Where M is the set of all training and test vectors for one test run and α is a variable to influence the number of clusters based on the size of M.

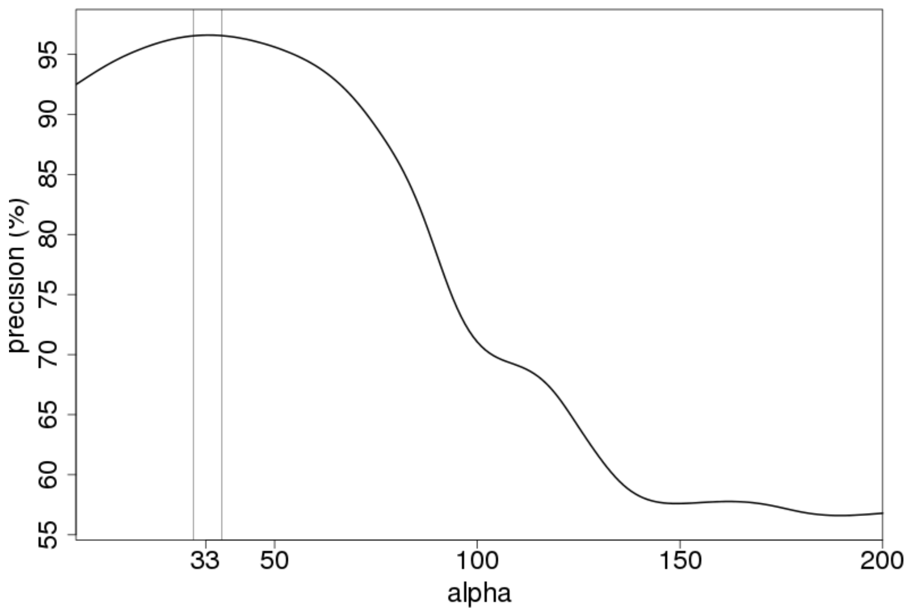


Figure 6.8: Clustering parameter evaluation.

The software was implemented in Python using the module "scipy" available from the SciPy.org project website [4]. As mentioned before, the training of the method was done by a c-means clustering algorithm into c clusters. After the calculations are complete, all clusters and their vectors are inspected. Similar to the analysis of a trained SOM, all clusters are suspicious that only contain test vectors. These vectors have to be different in some way, otherwise they would belong to a cluster that includes training vectors too. Thus all vectors belonging to suspicious clusters are marked.

The figure 6.7 displays a typical outlier situation. The clustering has been completed and four clusters were requested as output. The two large clusters c_2 and c_4 contain the majority of the vectors, the clusters c_1 and c_3 on the other hand are outliers. The main purpose of the clustering based outlier detection is to classify the elements from the clusters c_1 and c_3 as outliers such that they can be inspected further.

As described before, an essential part of the clustering algorithm is the desired amount of clusters that the result will contain. Formula 6.4 defined the number of clusters to be the number of elements divided by a factor α. The range of α is therefore at most $[1, |M|]$. The chosen outlier detection algorithm implies that a too large number of clusters will lead to many false positives where a too small number of clusters will give a high amount of false negatives. After some preliminary testing, the maximum value for α was defined to be 200. To find optimal values for α, all natural numbers from 1 to 200 were tested, giving the following set:

$$A = \{1, 2, \dots, 200\}$$

As test subject, the setup from section 5.1 was used. Testing all values of A requires a total of 200 test cases to evaluate the parameter α for the clustering method. The output of the test is

[4] http://www.scipy.org/ (last accessed: 29.09.2014)

again the percentage p of correctly identified test cases. The resulting vector of each test is a tuple $v = (\alpha, p)$.

The data set of result vectors is visualised in figure 6.8. Starting from the left the precision starts high and continues increasing until its peak at about $\alpha = 33$. Further on, the graph experiences a significant drop of precision with an α value higher than about 60. Following this suggestions it is safe to say α should be selected from the interval $[20, 50]$.

6.5.2 Results of the Evaluation

The evaluation of the parameter α gave a strong suggestion for the value selection. With this result the final test run of the clustering based outlier detection method can now be applied. The parameter was chosen to be $\alpha = 33$ and the clustering method was executed with this parameter. Since the used algorithm in the implementation is not a deterministic algorithm, several test runs have been executed. The statistics of the result are the following:

	mean		median		std. dev.	
correct	20576.70	96.53%	20566.50	96,49%	66.15	0.31%
false positive	620.00	2.90%	624.00	2.92%	46.22	0.22%
false negative	118.30	0.55%	140.00	0.65%	41.14	0.19%

Table 6.4: Evaluation result of the c-Means Clustering method.

7 Summary and Discussion of Evaluation Results

In the first part of this work, composed of chapter 2 and 3, explanations on the scientific background of this thesis were presented. The second part, consisting of the chapters 4, 5 and 6, introduced some new approaches on optimization of security test executions by using methods of computational intelligence. The following sections will give a detailed analysis of the evaluation results presented in chapter 6.

7.1 Results of the Evaluation

A first glance on the outcome of the machine learning method evaluation gives the impression that three of them produced adequate results. The accuracy of these three methods lie between 93.68% and 96.53%, which seems like acceptable values. However, the percentage of correctly identified test cases is not the only criterion in the overall evaluation. Therefore, all methods will now be compared to each other in detail regarding their evaluation results.

The Support Vector Machine scored about 93.7 percent points in accuracy. In detail this means, that out of 100 test cases, the SVM method identified about 94 cases correctly either as failure or no failure. Additionally, this particular method identified 5.79% of all test cases as outlier, but they were in fact none. Only 110 test cases or 0.52% of all cases that contained a flaw were not detected correctly and thereby not marked. This gives a relation of about 0.09 false negative cases per false positive $(1236 : 110 \approx 1 : 0.09)$.

The Self-Organizing Map identified about 95.7% of all test cases correctly, which is a slightly better result than the SVM. Considering the incorrectly identified cases, the SOM produced 3.49% false positive results and identified 0.78% failures of actually correct test cases. The relation between those two values is 0.22 false negative cases per false positive $(745 : 168 \approx 1 : 0.22)$.

Still on top of this is the clustering method with an arithmetic average of about 96.5% accuracy. The further results of this method yield a value of 2.90 percent points for test cases that were marked by the clustering algorithm, but had no flaw at all. In about 118 or 0.55% the method did not report anything suspicious but was wrong by doing so. This leads to a false positive:negative relation of 0.19 since $(620 : 118 \approx 1 : 0.19)$.

However, far behind those three lies the Artificial Neural Network with an arithmetic average of about 64.76% correctly identified cases. It identified about 17.28% of all cases incorrectly as failure and at about 17.94% of the test cases it did not detect the existing security flaw. This values leads to a ratio of about 1.03 with even more false negatives as false positives $(3684 : 3825 \approx 1 : 1.03)$.

An algorithm which has no information about the underlying test cases at all, would result in a purely random guessing algorithm. For every test case it would decide based on a random number, whether the test case is an outlier or no, resulting in an average accuracy of about 50%. Comparing the results of the ANN to this random algorithm, we can conclude that the network still returns much better results than this simple guess and check algorithm. However, 64.76% accuracy is by far not a practical value. Using this ANN implementation for example during an penetration test,

ML method	accuracy	false pos. (fp)	false neg. (fn)	ratio fp:fn
Artificial Neural Network	64.76%	17.28%	17.94%	1 : 1.03
Support Vector Machine	93.68%	5.79%	0.52%	1 : 0.09
Self-Organizing Map	95.71%	3.49%	0.78%	1 : 0.22
c-Means Clustering	96.53%	2.90%	0.55%	1 : 0.19

Table 7.1: All results of the ML method evaluation combined.

ML method	mean	median	std.dev
Artificial Neural Network	$17.38s$	$17.71s$	1.48
Support Vector Machine	$29.70s$	$30.30s$	4.18
Self-Organizing Map	$139.49s$	$147.61s$	18.61
c-Means Clustering	$35.33s$	$36.70s$	4.93

Table 7.2: The result of the running time evaluation of the ML methods.

the tester could not trust the results of the algorithm. With the knowledge that about one third of all results of the algorithm are incorrect, this would lead to a situation where every test case has to be double checked manually. Since the goal is to optimize the security test execution, this is not a desired situation.

In table 7.1 all the results of the evaluation phase are compiled together for an overview. Visible is the accuracy, the relative amount of false positives and false negatives, along with the ratio of the false positive compared to the false negative results. The entries are sorted by their accuracy value.

Having a low value of false negative means, by definition, that the program did only miss a few vulnerabilities in the test setup. In other words, it actually did find most of the vulnerabilities. A high value of false negative results on the other hand means that many vulnerabilities remained hidden from the program and thus also from the tester. With respect to security testing it seems more applicable to accept a slightly higher rate of false positives in trade of a smaller number of false negative results. In this scenario the tester maybe has to double check some of the test cases the method marked as outlier. However there will be a high probability that the program missed only a few security flaws. The other way around, with a high rate of false negative and a low rate of false positive test cases, the output of the program will be correct most of the time. Still, a probably very high number of vulnerabilities could not be discovered by the tool, making it a bad choice for security testing.

To gain some information on the time required from each method to execute a complete test run, an analysis on the running times was made. To gain statistical relevance, all prototypes have been executed 100 times in an encapsulated environment and statistical values have been extracted. The running times in seconds of all four method are displayed in table 7.2.

From this results we can conclude that three out of four algorithms, SOM, SVM and c-Means Clustering scored very high values. However, the results of this three methods are very close to each other. In the following sections the evaluation of all methods will be examined more carefully and more details will be given.

7.1.1 Support Vector Machine

With the help of the ”sklearn” module, the prototype implementation was done very fast in the language python. The whole algorithm was completely coded in about 20 lines, not counting comments. This included input data preparation as well as the output of all detected outliers. As mentioned in section 6.4, this code even included an iterative process of finding the smallest possible error ratio by stepwise decreasing the value for μ.

The average running time of about 30 seconds for a complete test over all $21,000$ test vectors is about as fast as the clustering method and much faster than the SOM prototype implementation. Assuming that a fixed value for μ can be found that is optimal or near-optimal over all cases, this would further improve the running time of the algorithm. Even the source code would lose some more complexity with the drop out of the parameter detection part.

The parameter evaluation for a value of the kernel coefficient displayed a very stable result over all chosen values of γ (see figure 6.6), leading to the conclusion that this parameter does not have a huge impact on the overall performance of the algorithm. Furthermore this leads to the save assumption that the SVM algorithm will produce stable results in various test environments.

The ratio between the false positive and the false negative results was calculated. The SVM presented itself with the lowest value compared to the other methods, with a ratio of 0.09.

The description of the prototype stated that in constructing the support vectors only actual training vectors have been used. This means that the program can build its support vectors as soon as the training phase is finished. It does not need to wait until the whole test run is completely executed. This situation may be advantageous regarding a possible implementation into the fuzzolution project. It opens the possibility to have instant feedback after each test case.

7.1.2 Self-Organizing Map

The implementation of the prototype was a more complex than that of the SVM since the program was split in two parts. The first part was done by a shell script and used the ”SOMToolbox’ application directly to grow and store the map. The second part was written in Java and used the framework of the toolbox to compare the input vectors to the existing SOM and deduce the outliers thereby.

Since for every one of the 76 test runs a completely new SOM had to be grown, it affected the running time of the program. As mentioned and discussed in section 6.3 a very large amount of training iterations were chosen. Both reasons combined lead to the rather long 140 seconds per complete test. However, due to this fact there is still a huge potential of optimizations that will improve the running time greatly.

The parameter evaluation of the size of the map indicated a preference for smaller maps (see figure 6.4). While this fact was sufficiently proven for the presented test environment, it may not be a valid assumption for an arbitrary test environment. This may lead to an uncertainty in the practical usage of the SOM method.

Regarding the ratio of the false positive and false negative results, the table 7.1 stated a value of 0.22 false negatives per false positive. Disregarding the ANN prototype, this is the highest ratio compared to the other methods. Compared to the SVM and clustering method, this SOM implementation will likely miss more vulnerabilities than the others.

The section about the design of the SOM implementation mentioned, that both training and testing data were used in training the map. This leads to the situation with less flexibility, since the results of each test case can only be determined after the whole test run is completed.

7.1.3 Clustering

The clustering prototype was implemented in python with the help of the ”scipy” module. This module simplified the source code greatly and the core part was done in about 15 lines of code. Already included in this count is data input and outlier output. The clustering program was by far the simplest program compared to the other three implementations.

Each evaluation test required a running time of about 35 seconds, which is comparable with the implementation of the SVM method. However, in this case there is not much optimization potential left besides switching to a different module or language.

The parameter evaluation for the number of clusters showed a clear peak at about $\alpha = 33$, displayed in figure 6.8. However, this value is not a constant fixing the number of clusters, but rather a part of a formula used in calculating it. The cluster calculation was defined in formula 6.4. As can be seen, another major part in this calculation is the number of the size of the input data. This leads to the conclusion that α may be assigned a fixed value but the number of clusters still may vary from case to case.

The ratio between the false positive and false negative test case results of the clustering method were calculated as 0.19. Thus, for every false positive the method produced, it also produced 0.19 false negative results. This ratio is slightly better, but still comparable to the value that was calculated for the SOM method.

7.1.4 Artificial Neural Network

As mentioned in section 6 the ANN prototype was implemented with the help of the Java framework ”Neuroph”. It offered great support in developing the network and training it. Still, the structure of the neural network and additionally the algorithm that detected the outliers had to be implemented by hand. The coding and testing of this program took quite some time. All together the source code consists of about 300 lines of Java code, not counting comments.

Although the source code was the longest of all prototypes, the running time of the outlier detection based on the artificial neural network still was very fast. On average, the program required about 17.5 seconds for a complete test run, which is about half as long as the faster of the other methods needed.

Looking at the parameter evaluation in figure 6.2 gives some clear statements about the outlier parameter α and the learning rate γ. The learning rate did not seem to have a lot of impact into the overall precision of the prototype. Only due to a slight increase in precision, a recommendation was given for the interval $[0.1, 0.4]$. The parameter α on the other hand indicated a high precision area inside the interval $[1.00, 1.03]$. However, there is no evidence that both parameter selections will produce stable results in a changing environment. This could either lead to a further drop of precision or possible even an increase in precision if tested against a different SUT. Without further investigation it is not possible to make any predictions on this matter.

An advantage of the neural network design is its flexibility regarding the training and learning phase of a test run. As mentioned in the prototype design descriptions in chapter 6 the network is trained exclusively with training vectors. During the testing phase, each testing vector can be applied to the network directly and conclusions about it being an outlier or not can be drawn at once.

Another shortcoming of the ANN is the ratio between the false positives and false negatives it displayed. About as many test cases without a vulnerability have been marked as outliers as test cases with vulnerabilities have not been marked.

	SVM	SOM	Clustering	ANN
detection accuracy	93.58%	95.71%	**96.53%**	64.76%
source code complexity	**simple**	complex	**simple**	complex
running time	30s	140s	35s	**18s**
parameter stability	**stable**	unknown	**stable**	unknown
training flexibility	**flexible**	fixed	fixed	**flexbile**
ratio $fp{:}fn$	1 : **0.09**	1 : 0.22	1 : 0.19	1 : 1.03

Table 7.3: Summary of all results of the ML method evaluation.

7.2 Discussion of Results

To give a better overview of the results, the data from the detailed analysis of the evaluation result in section 7.1 is compiled and presented in table 7.3. The columns represent each of the four methods: Support Vector Machine, Self-Organizing Map, c-Means Clustering and Artificial Neural Network. The rows of the table contain information about all investigated properties of the implementations. Highlighted are those columns with the most promising results. The row *detection accuracy* reflects the precision of each method in finding vulnerabilities as percentage of correctly identified test cases. *Source code complexity* contains one of the two values *simple* or *complex* and indicates the overall complexity of the source code of the implementation as described before. The *running time* is given in seconds of run time required for a complete test run. The row *parameter stability* summarises the considerations on the parameter evaluation for each method. The term *stable* indicates that the implementation with the selected parameter will produce similar results in an arbitrary test environment. If no conclusion could be drawn to predict the stability of the implementation, then the term *unknown* is present. If an implementation grants some flexibility regarding the training and testing phase this is displayed in the row *training flexibility*. A flexible method has the property that the training of the machine learning method can be done right after the training phase of the fuzzing run is complete. Afterwards, for each test vector produced, conclusions can immediately be drawn about its state as outlier. Such a situation is indicated by the term *flexible*. The term *fixed* is present if determination of test vectors can only be done after a completed test run. Finally, the row *ratio fp:fn* is a comparison of the ratios between the false positive and the false negative results of each method.

These results indicate clearly, that the implementation of the Artificial Neural Network is not competitive in its current state. A lot of effort was put into this method, much more compared to the other three prototypes, yet it was not able to produce any comparable results. Some investigations on the issue revealed that the training of the network did sometimes call on some strange behaviour in the network. This was even the case when the training was successful in a mathematical way, meaning that the mean-squared error of the network was below the required value of 0.01. Still, the network was not able to detect any outliers properly in the given set of test vectors. This situation could occur at any test run regardless of the actual data it contained. Further, due to the non-deterministic training of the network this behaviour did only occur occasionally at a fixed test run.

The Self-Organizing Map method did show some minor disadvantages compared to the SVM and clustering method. One is the need to create a new SOM for each test run which leads to a slightly higher running time. However, regarding the total security test process with test preparation and manual analysis phase, the required time of the SOM method can be considered as being insignificant. The uncertainty on the map size and its inflexible training algorithm are some additional minor drawbacks. Nevertheless it is undeniable, that this method produced formidable results in the detection of outliers.

The two candidates Support Vector Machine and clustering method stand out with a simple source code, acceptable running time and stable parameter conditions. Additionally, both scored comparable and high results in the evaluation of the vulnerability detection accuracy. The clustering method did produce slightly better result of about three percent points. On the other hand it does not offer the training flexibility of the SVM method and its false positive/negative ratio is slightly worse.

Taking all results of this evaluation into account it is obvious that every method has some advantages and disadvantages. This conclusion is in line with the famous ”No-Free-Lunch”-Theorem formalized by Wolpert [87] and later specified to machine learning algorithms [88]. This theorem states that over all possible functions to be learned, the average performance of all learning algorithms is equivalent.

The implementation and evaluation of the prototypes has shown that the test quality is highly dependent on a wide array of different parameters. To gain optimal results, it is important to have an automated determination of the parameters used by the method. A lot of scientific work has been dedicated to this task already, for example by Murata [58] and Luk [50] for the ANN, by Liu [49] and Kumar [43] for the SOM, by Min [55] and Wu [89] for the SVM and by Ester [19], McDonald [53] and Brohée [13] for clustering algorithms.

Another very important aspect of the performance of machine learning tools in the context of automated security scanning is the architecture of the combined system. The quality of the security testing tool depends on the quality of several components and one crucial component is the Analyze Manager. As described in chapter 4, the machine learning methods get their input in form of the System Behaviour Model. By its definition (see formula 4.1), this model is a list of vectors extracted from all analyzers used in the test setup. Thus the quality of the System Behaviour Model depends mainly on two circumstances: on the quality of each analyzer and additionally on the number of used analyzers. It is important for the quality of the model that the output of each analyzer reflects the actual behaviour of the tested system as exactly as possible. In addition, the number of used analyzers is an important part, too. The more of them are used, the higher is the probability of capturing a behaviour that reveals a vulnerability in the SUT. For example, only an analyzer measuring the response time can detect a timed SQL injection attack. If no such analyzer is contained in the test setup, these types of injection could not be detected at all. Following these statements, it is a highly important task to maintain a large set of high quality analyzers.

The selection of attack vectors is another important part in the automated security testing process. In order to design significant test cases, it is important to have a good coverage of all possible attack vectors (see section 4.2), since the probability of finding flaws can only increase with the amount of input values sent. Without some additional information this can quickly become an intractable problem, leading to drawbacks of either poor test coverage or immense run time. Therefore, it is important to have a high quality set of attack strings that can be used by the security scanner.

A possible drawback regarding the evaluation of the methods was mentioned in section 7.1. By the evaluation of the prototype implementations it has already been sufficiently proven, that the two most suitable methods are ideal candidates for the integration into a security testing framework. However, the evaluation was mainly based on tests against web applications. These types of applications are a very important part of software products, however the range of product types is far bigger. The selected methods should be tested against several different application types and protocols. The results of these tests will help to further confirm the generality of the methods presented in this thesis.

8 Conclusion

Up to now, most of the analysis of an automated security test execution had to be done by manually checking a huge amount of test cases. This is a very time consuming process that limits the overall usefulness and efficiency of automated testing. The goal of this thesis was to present an approach to significantly improve the analysis phase by using methods of artificial intelligence. These methods should detect outliers in the behaviour of the tested system and thus greatly reduce the amount of test cases that have to be checked manually.

This thesis presented an approach that automatically detects vulnerabilities during an automated security test. The approach is based on the idea of learning the behaviour of the SUT and building a System Behaviour Model by sending functional test cases. After the learning phase, simulated attacks are executed which can trigger unexpected behaviour in the SUT. The behaviour of the SUT is captured and checked against the learned model representing normal system conditions. Any difference in the behaviour of the system during the attack simulation can be caused by a security failure inside the SUT.

The automatically built System Behaviour Model consists of the behaviour data of the SUT generated by functional test cases and is used to train a machine learning algorithm. During the attack simulation, the captured behaviour of the SUT is given to the algorithm. It compares the data to the learned model and automatically classifies the behaviour as normal or abnormal.

This approach was already presented to the scientific community, peer-reviewed and published in March 2013 as ”Generic Approach for Security Error Detection Based on Learned System Behavior Models for Automated Security Tests” by C. Schanes, A. Hübler, F. Fankhauser and T. Grechenig [73].

This thesis further presented an evaluation of four machine learning methods for their use in automated security testing: Artificial Neural Networks, Self-Organizing Maps, Support Vector Machine and c-Means Clustering. To evaluate the methods, prototypes have been implemented and tested against the specialised vulnerability scanner evaluation platforms WAVSEP and ZAPWAVE.

The results presented in the previous chapter showed that three of the investigated prototypes are well suited solutions to the problem addressed by this thesis. These prototypes implemented the methods: Self-Organizing Map, Support Vector Machine and c-Means Clustering. The results showed that a high vulnerability detection rate could be reached by these implementations in the test environment.

All prototype implementations have been evaluated by testing them against two different platforms. Both platforms contain several applications with or without vulnerabilities. However, all these applications are based on websites and are accessible via HTTP. These circumstances limit the significance of the evaluation results slightly. Without further investigations it remains unknown if the evaluation will have the same results in an arbitrary environment. For example, if the prototype implementations were tested against a SUT communicating with some other protocol, like the Session Initiation Protocol (SIP) used for Voice over Internet Protocol (VoIP) applications.

During the testing process of the prototypes, a lot of time was invested in the evaluation of the optimal parameters of each prototype. The chosen parameters based on this evaluation are well tested in the test environment and returned good results. For a future work it would be better to use an automated approach of parameter selection. This would reduce another uncertainty in the generalised results of the prototypes and in addition some amount of manual labor for each test environment.

Machine learning methods can support the error detection capabilities of automated security tests. Integrating these methods in a security testing framework can reduce the manual work required for the analysis phase. Due to the automated detection of abnormal behaviour in the SUT, the manual analysis will mainly consist of checking the test cases corresponding to the detected behaviour. With the reduction of manual labor needed for executing the test, the overall efficiency of automated testing will improve.

Bibliography

[1] H. Abdelnur et al. *Spectral Fuzzing: Evaluation & Feedback*. Tech. rep. Feb. 2010, p. 40.

[2] S. Abe. *Support Vector Machines for Pattern Classification*. Advances in Computer Vision and Pattern Recognition. Springer, 2005. ISBN: 9781852339296.

[3] E. Alpaydin. *Introduction to machine learning*. Adaptive computation and machine learning. MIT Press, 2010. ISBN: 9780262012430.

[4] T. Ambwani. "Multi class support vector machine implementation to intrusion detection". In: *Neural Networks, 2003. Proceedings of the International Joint Conference on*. Vol. 3. IEEE. 2003, pp. 2300–2305.

[5] R. J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2010. ISBN: 9781118008362.

[6] M. F. Augusteijn and B. A. Folkert. "Neural network classification and novelty detection". In: *International Journal of Remote Sensing* 23.14 (2002), pp. 2891–2902.

[7] X. Bao, T. Xu, and H. Hou. "Network Intrusion Detection Based on Support Vector Machine". In: *International Conference on Management and Service Science*. 2009. DOI: 10.1109/ICMSS.2009.5304051.

[8] S. Bekrar et al. "Finding Software Vulnerabilities by Smart Fuzzing". In: *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. Mar. 2011, pp. 427 –430.

[9] J. C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms*. Norwell, MA, USA: Kluwer Academic Publishers, 1981. ISBN: 0306406713.

[10] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003. ISBN: 9780201440997.

[11] M. Bishop. *Introduction to Computer Security*. Addison-Wesley, 2005. ISBN: 9780321247445.

[12] G. Bossert, G. Hiet, and T. Henin. "Modelling to Simulate Botnet Command and Control Protocols for the Evaluation of Network Intrusion Detection Systems". In: *Conference on Network and Information Systems Security (SAR-SSI)*. May 2011, pp. 1 –8.

[13] S. Brohee and J. van Helden. "Evaluation of clustering algorithms for protein-protein interaction networks". In: *BMC bioinformatics* 7.1 (2006), p. 488.

[14] G. Chen and T. T. Pham. *Introduction to Fuzzy Sets, Fuzzy Logic and Fuzzy Control Systems*. Taylor & Francis, 2001. ISBN: 9780849316586.

[15] Gavrilut D. et al. "Malware detection using machine learning". In: *International Multiconference on Computer Science and Information Technology - IMCSIT*. 2009, pp. 735–741. DOI: 10.1109/IMCSIT.2009.5352759.

[16] E. W. Dijkstra. "The humble programmer". In: *Commun. ACM* 15.10 (1972), pp. 859–866. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/355604.361591.

[17] P. Engebretson. *The Basics of Hacking and Penetration Testing: Ethical Hacking and Penetration Testing Made Easy*. Syngress basics series. Elsevier Science, 2011. ISBN: 9781597496568.

[18] W. Ertel. *Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung*. Computational Intelligence. Vieweg+Teubner Verlag, 2009. ISBN: 9783834807830.

[19] M. Ester et al. “A density-based algorithm for discovering clusters in large spatial databases with noise.” In: *Kdd*. Vol. 96. 1996, pp. 226–231.

[20] R. Ettisberger. *IT Security & Hacking*. Norderstedt: Books on Demand GmbH, 2006.

[21] BlackHat Conference Europe. *Speaker Description*. (last accessed: 29.09.2014). 2013. URL: https://www.blackhat.com/eu-13/speakers/Shay-Chen.html.

[22] I. Firdausi et al. “Analysis of machine learning techniques used in behavior-based malware detection”. In: *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on*. IEEE. 2010, pp. 201–203.

[23] P. Godefroid, A. Kiezun, and M. Y. Levin. “Grammar-based whitebox fuzzing”. In: *ACM Sigplan Notices*. Vol. 43. 6. ACM. 2008, pp. 206–215.

[24] P. Godefroid, M. Y. Levin, and D. A. Molnar. “Automated Whitebox Fuzz Testing”. In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. 2008.

[25] A. K. Gosh and A. Schwartzbard. “A Study in Using Neural Networks for Anomaly and Misuse Detection”. In: *USENIX Security Symposium*. 1999.

[26] C. Han et al. “An intrusion detection system based on neural network”. In: *Mechatronic Science, Electric Engineering and Computer (MEC), 2011 International Conference on*. Aug. 2011, pp. 2018 –2021.

[27] S. S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall International Editions Series. Prentice Hall, 1999. ISBN: 9780139083853.

[28] Z. He, X. Xu, and S. Deng. “Discovering cluster-based local outliers”. In: *Pattern Recognition Letters* 24.9 (2003), pp. 1641–1650.

[29] D.O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. Taylor & Francis Group, 2002. ISBN: 9780805843002.

[30] W. C. Hetzel. *The complete guide to software testing*. QED Information Sciences, 1988. ISBN: 9780894352423.

[31] A. J. Hoglund, K. Hätonen, and A. S. Sorvari. “A computer host-based user anomaly detection system using the self-organizing map”. In: *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*. Vol. 5. IEEE. 2000, pp. 411–416.

[32] W. Hu, Y. Liao, and V. R. Vemuri. “Robust Support Vector Machines for Anomaly Detection in Computer Security.” In: *ICMLA*. 2003, pp. 168–174.

[33] N. Hubballi, S. Biswas, and S. Nandi. “Fuzzy mega cluster based anomaly network intrusion detection”. In: *Network and Service Security, 2009. N2S’09. International Conference on*. IEEE. 2009, pp. 1–5.

[34] Alshanetsky. I. *php - architect’s Guide to PHP Security*. Tabini & Associates, Sept. 2005.

[35] D. Ippoliti and X. Zhou. “An adaptive growing hierarchical self organizing map for network intrusion detection”. In: *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*. IEEE. 2010, pp. 1–7.

[36] K. A. Jalili, M. H. Kamarudin, and M. N. Masrek. “Comparison of Machine Learning algorithms performance in detecting network intrusion”. In: *International Conference on Networking and Information Technology*. 2010. DOI: 10.1109/ICNIT.2010.5508526.

[37] H. Jiang and X. Zhao. “Study on the Network Intrusion Detection Model Based on Genetic Neural Network”. In: *Modelling, Simulation and Optimization, 2008. WMSO ’08. International Workshop on*. Dec. 2008, pp. 60 –64.

[38] M. F. Jiang, S. S. Tseng, and C. M. Su. “Two-phase clustering process for outliers detection”. In: *Pattern recognition letters* 22.6 (2001), pp. 691–700.

[39] M. A. B. Junior, F. B. de Lima Neto, and J. C. S. Fort. “Improving black box testing by using neuro-fuzzy classifiers and multi-agent systems”. In: *Hybrid Intelligent Systems (HIS), 2010 10th International Conference on*. Nov. 2010, pp. 25 –30.

[40] L. Khan, M. Awad, and B. Thuraisingham. “A new intrusion detection system using support vector machines and hierarchical clustering”. In: *The VLDB Journal*. Vol. 16. 4. 2007, pp. 507 –521.

[41] T. Kiziloren and E. Germen. “Network traffic classification with Self Organizing Maps”. In: *Computer and information sciences, 2007. iscis 2007. 22nd international symposium on*. Nov. 2007.

[42] T. Kohonen. *Self-Organizing Maps*. Springer Series in Information Sciences Series. Springer-Verlag GmbH, 2001. ISBN: 9783540679219.

[43] G. S. Kumar, P. K. Kalra, and S. G. Dhande. “Curve and surface reconstruction from points: an approach based on self-organizing maps”. In: *Applied Soft Computing* 5.1 (2004), pp. 55–66.

[44] L. M. Laird and C. Brennan. *Software measurement and estimation: a practical approach*. John Wiley & Sons, 2006. ISBN: 9780471676225.

[45] C. E. Landwehr et al. “A taxonomy of computer program security flaws”. In: *ACM Computing Surveys* 26 (3 1994), pp. 211–254. DOI: 10.1145/185403.185412.

[46] P. Lichodzijewski, A. N. Zincir-Heywood, and M. I. Heywood. “Dynamic intrusion detection using self-organizing maps”. In: *The 14th Annual Canadian Information Technology Security Symposium (CITSS)*. Citeseer. 2002.

[47] O. Linda, T. Vollmer, and M. Manic. “Neural network based intrusion detection system for critical infrastructures”. In: *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*. IEEE. 2009, pp. 1827–1834.

[48] M. A. van der Linden. *Testing Code Security*. Taylor & Francis, 2007. ISBN: 9781420013795.

[49] Y. Liu, R. H. Weisberg, and C. N. K. Mooers. “Performance evaluation of the self-organizing map for feature extraction”. In: *Journal of Geophysical Research: Oceans (1978–2012)* 111.C5 (2006).

[50] K. C. Luk, J. E. Ball, and A. Sharma. “A study of optimal model lag and spatial inputs to artificial neural network for rainfall forecasting”. In: *Journal of Hydrology* 227.1 (2000), pp. 56–65.

[51] H. Lukashevich, S. Nowak, and P. Dunker. “Using one-class SVM outliers detection for verification of collaboratively tagged image training sets”. In: *Multimedia and Expo, 2009. ICME 2009. IEEE International Conference on*. IEEE. 2009, pp. 682–685.

[52] W. S. McCulloch and W. Pitts. “A logical calculus of the ideas immanent in nervous activity”. English. In: *The bulletin of mathematical biophysics* 5 (4 1943), pp. 115–133. ISSN: 0007-4985. DOI: 10.1007/BF02478259.

[53] A. B. McDonald and T. F. Znati. “A mobility-based framework for adaptive clustering in wireless ad hoc networks”. In: *Selected Areas in Communications, IEEE Journal on* 17.8 (1999), pp. 1466–1487.

[54] J. M. Mendel and R. W. McLaren. “8 Reinforcement-Learning Control and Pattern Recognition Systems”. In: *Adaptive, Learning and Pattern Recognition Systems Theory and Applications*. Vol. 66. Mathematics in Science and Engineering. Elsevier, 1970, pp. 287 –318.

[55] J. H. Min and Y. C. Lee. “Bankruptcy prediction using support vector machine with optimal choice of kernel function parameters”. In: *Expert systems with applications* 28.4 (2005), pp. 603–614.

[56] S. Mukkamala, G. Janoski, and A. Sung. “Intrusion detection using neural networks and support vector machines”. In: *Neural Networks, 2002. IJCNN’02. Proceedings of the 2002 International Joint Conference on*. Vol. 2. IEEE. 2002, pp. 1702–1707.

[57] S. A. Mulay, P. R. Devale, and G. V. Garje. “Decision tree based Support Vector Machine for Intrusion Detection”. In: *Networking and Information Technology (ICNIT), 2010 International Conference on*. June 2010, pp. 59 –63.

[58] N. Murata, S. Yoshizawa, and S. I. Amari. “Network information criterion-determining the number of hidden units for an artificial neural network model”. In: *Neural Networks, IEEE Transactions on* 5.6 (1994), pp. 865–872.

[59] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. ITPro collection. Wiley, 2011. ISBN: 9781118133156.

[60] H. Q. Nguyen. *Testing Applications on the Web: Test Planning for Internet-Based Systems*. Wiley, 2001. ISBN: 9780471437642.

[61] S. H. Oh and W. S. Lee. “An anomaly intrusion detection method by clustering normal user behavior”. In: *Computers & Security* 22.7 (2003), pp. 596–612.

[62] S. H. Oh et al. “Anomaly Intrusion Detection Based on Clustering a Data Stream”. In: *Information Security*. Ed. by SokratisK. Katsikas et al. Vol. 4176. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 415–426. ISBN: 978-3-540-38341-3. DOI: 10.1007/11836810_30.

[63] N. H. Park, S. H. Oh, and W. S. Lee. “Anomaly intrusion detection by clustering transactional audit streams in a host computer”. In: *Information Sciences* 180.12 (2010), pp. 2375–2389.

[64] C. P. Pfleeger and S. L. Pfleeger. *Security in Computing*. Prentice Hall professional technical reference. Prentice Hall PTR, 2006. ISBN: 9780130355485.

[65] L. Portnoy. “Intrusion detection with unlabeled data using clustering”. In: (2000).

[66] S. T. Powers and J. He. “A hybrid artificial immune system and Self Organising Map for network intrusion detection”. In: *Information Sciences* 178.15 (2008), pp. 3024 –3042.

[67] Open Web Application Security Project. *Top 10 2013*. (last accessed: 29.09.2014). 2013. URL: https://www.owasp.org/index.php/Top_10_2013.

[68] J. Radatz, A. Geraci, and F. Katki. “IEEE standard glossary of software engineering terminology”. In: *IEEE Std* (1990).

[69] M. Ramadas, S. Ostermann, and B. Tjaden. “Detecting anomalous network traffic with self-organizing maps”. In: *Recent Advances in Intrusion Detection*. Springer. 2003, pp. 36–54.

[70] K. Rieck et al. “Automatic analysis of malware behavior using machine learning”. In: *Journal of Computer Security* 19.4 (2011), pp. 639–668.

[71] T.J. Ross. *Fuzzy Logic with Engineering Applications*. Wiley, 2009. ISBN: 9780470748510.

[72] S.J. Russell and S.R.P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2010. ISBN: 9780136042594.

[73] C. Schanes et al. “Generic Approach for Security Error Detection Based on Learned System Behavior Models for Automated Security Tests”. In: *Proceedings of the Sixth IEEE International Conference on Software Testing, Verification and Validation* (2013), pp. 453–460.

[74] C. Schanes et al. “Security Test Approach for Automated Detection of Vulnerabilities of SIP-based VoIP Softphones”. In: *International Journal On Advances in Security* 4.1 and 2 (Sept. 2011), pp. 95–105.

[75] B. Schneier. *Secrets & Lies*. Indianapolis, Indiana: Wiley Publishing, Inc., 2004.

[76] R. W. Shirey. *Security Architecture for Internet Protocols: A Guide for Protocol Designs and Standards*. Internet Draft: draft-irtf-psrg-secarch-sect1-00.txt. 1994.

[77] J. Shun and H. A. Malki. “Network Intrusion Detection System Using Neural Networks”. In: *Natural Computation, 2008. ICNC ’08. Fourth International Conference on*. Vol. 5. Oct. 2008, pp. 242 –246.

[78] E. Skoudis and T. Liston. *Counter Hack Reloaded*. Pearson Education, Inc., 2006.

[79] I. Sommerville. *Software Engineering*. Pearson Education, 2011. ISBN: 9780133001495.

[80] Q. Song, W. Hu, and W. Xie. “Robust support vector machine with bullet hole image classification”. In: *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 32.4 (2002), pp. 440–448.

[81] A. Takanen, J. D. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008. ISBN: 978-1-59693-214-2.

[82] H. H. Thompson. “Why security testing is hard”. In: *Security Privacy, IEEE* 1.4 (2003), pp. 83–86. ISSN: 1540-7993. DOI: 10.1109/MSECP.2003.1219078.

[83] H. H. Thompson and J. A. Whittaker. “Testing for Software Security”. In: (2002). URL: http://www.drdobbs.com/184405196.

[84] H. Ting, W. Yong, and T. Xiaoling. “Network traffic classification based on Kernel Self-Organizing Maps”. In: *Intelligent Computing and Integrated Systems (ICISS), 2010 International Conference on*. Oct. 2010, pp. 310 –314.

[85] J. Watkins and S. Mills. *Testing IT: An Off-the-Shelf Software Testing Process*. Cambridge University Press, 2010. ISBN: 9780521148016.

[86] C. Willems, T. Holz, and F. Freiling. “Toward Automated Dynamic Malware Analysis Using CWSandbox”. In: *IEEE Secur Priv* 5.2 (2007), pp. 32–39.

[87] D. H. Wolpert. “The lack of a priori distinctions between learning algorithms”. In: *Neural computation* 8.7 (1996), pp. 1341–1390.

[88] D. H. Wolpert. “The supervised learning no-free-lunch theorems”. In: *Soft Computing and Industry*. Springer, 2002, pp. 25–42.

[89] C. H. Wu et al. “A real-valued genetic algorithm to optimize the parameters of support vector machine for predicting bankruptcy”. In: *Expert systems with applications* 32.2 (2007), pp. 397–408.

[90] J. Zhao, M. Chen, and Q. Luo. “Research of intrusion detection system based on neural networks”. In: *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*. May 2011, pp. 174 –178.

[91] T. Zhou and L. Yang. “The research of intrusion detection based on genetic neural network”. In: *Wavelet Analysis and Pattern Recognition, 2008. ICWAPR ’08. International Conference on*. Vol. 1. Aug. 2008, pp. 276 –281.