Informatics

Efficiency of Security Test Concepts Exemplified by a Decentralised Component in a Large IT Infrastructure With High Protection Needs in the eHealth Sector

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

René Czerny
Matrikelnummer 00825750

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuer: Thomas Grechenig
Mitwirkung: Florian Fankhauser

Wien, 4. Mai 2020

Unterschrift Verfasser

Unterschrift Betreuer

Informatics

Efficiency of Security Test Concepts Exemplified by a Decentralised Component in a Large IT Infrastructure With High Protection Needs in the eHealth Sector

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

René Czerny
Registration Number 00825750

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig
Assistance: Florian Fankhauser

Vienna, 4th May, 2020

Signature Author Signature Advisor

Efficiency of Security Test Concepts Exemplified by a Decentralised Component in a Large IT Infrastructure With High Protection Needs in the eHealth Sector

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

René Czerny
Matrikelnummer 00825750

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuer: Thomas Grechenig
Mitwirkung: Florian Fankhauser

Wien, 4. Mai 2020

Erklärung zur Verfassung der Arbeit

René Czerny

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.

Wien, 4. Mai 2020

René Czerny

Acknowledgements

I wish to thank the various people who helped me and supported me during the course of the creation of this work.

First, my parents Franz and Krystyna, who laid the foundation of my achievements years ago by raising me with love and care, and pointing me in the right direction when guidance was needed.

Special thanks goes to my sister Alina and brother Oliver who, with their unconditional support, motivated me to persevere whilst writing this thesis.

I want to thank my girlfriend Saša Šunjić-Alić for her constant encouragement, support and understanding when I spent days on finishing this work.

Finally, my thanks go to Thomas Grechenig and Florian Fankhauser for their endless patience, advice, and guidance, which made this possible in the first place.

'We shall not cease from exploration, and the end of all our exploring will be to arrive where we began and to know the place for the first time.'

T. S. Eliot

Kurzfassung

IT-Systeme gehören zum Kern vieler Geschäftsmodelle und sind aus dem Geschäftsalltag nicht mehr wegzudenken. Mit der Verwendung dieser IT-Systeme wird auch der IT-Sicherheit eine größere Bedeutung zugesprochen. Damit ist die Notwendigkeit umfassender Sicherheitstests eine logische Konsequenz - insbesondere in den Bereichen E-Government und E-Health, in denen hochsensible Daten verarbeitet werden. Es liegt in der Verantwortung jedes Dienstanbieters, die Maßnahmen zur Sicherheit der verarbeiteten Daten umzusetzen. Mit neuen und strengeren Datenschutzgesetzen, z.B. der EU-weit gültigen Datenschutz-Grundverordnung, wurden die Vorschriften zur Verarbeitung und Speicherung sensibler Daten, wie z. B. Gesundheitsdaten, verschärft. Die Strafen für Mängel an der Sicherheit solcher Systeme wurden signifikant erhöht und können schwerwiegende wirtschaftliche Schäden verursachen. Erhöhte Strafen gibt es im Falle von sensiblen Daten.

Dies macht Sicherheitstests für Systeme, die sensible Daten verarbeiten, unabdingbar. Da die Sicherheit eines Systems nicht bewiesen werden kann, sondern nur seine Unsicherheit, besteht die Notwendigkeit, sinnvolle Einschränkungen für eine effiziente Testdurchführung zu treffen. Diese Testgrenzen sollen sicherstellen, dass sowohl Budgetgrenzen eingehalten werden können als auch eine hinreichende Testabdeckung zur Prüfung der Systemsicherheit erreicht werden kann. In dieser Arbeit wird analysiert, wie ein Sicherheitstestkonzept unter Berücksichtigung des State-of-the-Art und Best Practices zur Reduktion von Testsuiten, der Automatisierung von Testfällen und der Priorisierung von Testfällen hinsichtlich Effizienz optimiert werden kann. Das Ergebnis ist eine Optimierungsmethode, mit deren Hilfe Sicherheitstestkonzepte effizienter gestaltet werden können.

Die vorgestellte Methode wurde an einem realen Beispiel, einem komplexen System mit sehr hohen Sicherheitsanforderungen im Bereich der elektronischen Gesundheitsdienste, evaluiert.

Schlüsselwörter

Sicherheitstestkonzept-Optimierung, Test-Suite-Reduzierung, Testfall-Automatisierung, Testfall-Priorisierung, e-health.

Abstract

IT systems are part of many business models in today's business world. With the question of security of these systems, IT security becomes a concern of growing importance. The need for extensive security testing is a logical consequence, especially when it comes to the domains of eGovernment and e-health, where highly sensitive data is handled. It is every service provider's responsibility to implement adequate security measures for the data's safety. With new data protection laws, e.g. the EU-wide general data protection laws, regulations regarding the processing and storage of sensitive data, like health data, became stricter and penalties for improper securing of the systems that handle such data are significant and can cause serious economic damage.

This results in the need for security testing. As the security of a system cannot be proven, as stated by Geer[34], but only its insecurity, this bears the necessity for constraints to provide efficient test execution. These constraints ensure that budget limits are not exceeded whilst maintaining sufficient test coverage.

This work analyses how a security test concept can be optimised for economic efficiency, taking into account the state-of-the-art and best practices in security testing, test suite reduction, test case automation, and test case prioritisation. The result is an optimisation method, which can help to optimise security test concepts.

The method was tested on a real-world example – a complex system with very high protection needs in the field of e-health.

Keywords

security test concept optimisation, test suite reduction, test case automation, test case prioritisation, e-health.

Contents

CHAPTER 1

Introduction

This work will present issues with security test efficiency, layout basics of IT security, security testing, and test optimisation and present a method to increase a security test's efficiency. The current section 'Introduction' will outline the problem this work tries to solve, the motivation behind it and the objective of this work. Finally, an orientation is given by presenting the structure of this thesis.

1.1 Problem Description

With IT security being a concern of growing importance, the need for extensive security testing is a logical consequence. This is especially the case when it comes to the domains of eGovernment and eHealth, where highly sensitive data is handled. It is every service provider's responsibility to make sure the data handled is safe. With new data protection laws, e.g. the EU-wide general data protection laws [67], regulations regarding the processing and storage of sensitive data, like health data, became stricter and penalties for improper securing of the systems that handle such data are significant and can cause serious economic damage.

In security testing, the tester can never be sure that the System Under Test (SUT) has no security vulnerabilities, as there always might be vulnerabilities, which have not been found yet. The challenge is to apply constraints to support efficient test execution, whilst ensuring sufficient coverage of the SUT. This raises the question on how security test concepts can be optimised for efficiency, to maximise the output within the given constraints.

For a case study, this work will use a complex system with very high protection need, which includes a VPN gateway and firewall, acts as a decentralised access component and establishes a secure connection to a central eHealth infrastructure. Its task is to make sure the data is processed properly and securely gets to the intended destination without being altered or intercepted/read by a third party.

The complexity of the aforementioned component requires meticulous analysis of possible attack vectors, given the current state-of-the-art in methods of vulnerability exploitation and the definition of a minimum level of security, the system needs to meet. Each additional vector can have side effects and influence other vectors to a certain degree, which as a consequence leads to exponential growth in the number of necessary test cases in the security test.

1.2 Motivation

The security of a system cannot be proven, as stated by Geer[34], because this would require the tester to know and enumerate all the possible security issues. Therefore, in the field of security testing, the tester works on proving a system's insecurity, whereas in regular functional testing the correct functionality is tested for. This bears the risk that without constraints, a security tester could endlessly test on a part of the SUT, going ever deeper into the matter, without testing the other parts of the SUT in the boundaries of a given testing budget. This leads to the issue that given the same cost, the benefit is reduced, as the SUT has not been sufficiently covered.

To prevent this, a security test concept helps to guide the tester in his journey through the system. To maximise the test coverage in the given budget, it is necessary to economically optimise this security test concept.

1.3 Objective

This work focuses on how a security test concept can be optimised for economic efficiency while preserving a sufficient level of IT security. The actual implementation and execution of the security test itself, as well as security tests that go beyond the borders of the component, are not part of this work.

The objective will be an analysis on how to optimise a security test concept, taking into account the state-of-the-art and best practices in security testing and to create a minimal test set, while maximizing the economic efficiency of the applied tests.

A thorough analysis of possible attack vectors that could pose a threat for data security includes the test object's scope and covers the whole system accurately. The number of test cases required to reach a significant amount of test coverage increases with the component's complexity. If the additional test cases can't be executed automatically, the increased number results in an even higher rise of the security test's execution time and cost. The suggested optimisation steps will make the security test easier and cheaper to execute, without losing test coverage.

The method itself will be tested and verified properly on a real-world component, which will provide data regarding its effectiveness to economically optimise a given test suite.

The presented form will be steps of an optimisation method, applicable to the previously created test set.

1.4 Structure of This Thesis

The work will start with laying the theoretical foundation needed to understand thoughts and theories in later chapters. Based on this foundation, the author will analyse different optimisation approaches, test them in practice and finish with a conclusion on security test optimisation. The following points will describe the methodology in detail:

- The first part consists of a discussion about the basic information needed to understand the outlined concepts in the later chapters. Extensive literature research will help to establish a firm theoretical base and explore the state-of-the-art in topics of software testing, software security, security testing in general. Furthermore, optimisation approaches that can be applied to the decentralised component's test, that lay the foundation of this work, will be elaborated.
- After establishing the foundation of this work, a theory on how a security test optimisation concept could look like will be discussed. A summary and best practice of the previous findings are presented and ready to be validated and tested in practice.
- To validate the theoretical findings, different criteria are established to conduct before/after comparisons. Before the described optimisation method can be tested on the real-world example, a complex component in the eHealth sector, the SUT and its subcomponents will be explored and a suite of security tests described. The established theory is then tested in practice and applied to the test suite of the SUT. The result will provide information on how effectively the steps optimise a complex component's security tests.
- Finally, the conclusion sums up the findings and gives an outlook for possible future research.

CHAPTER 2

Basic Information & Concepts

Section 2.1 will introduce basic concepts and methods of information security. Section 2.2 outlines the basics of software testing and serves as basis for Section 2.3 about penetration testing. Section 2.4 will introduce the topic of eHealth and, finally, Section 2.5 will explain the meaning, characteristics and issues of large IT infrastructures.

2.1 Information Security

There are several authorities in the field of information security, all using slightly different definitions of the term 'information security'. Examples thereof are as follows.

The National Institute of Standards and Technology (NIST) of the U.S. Department of Commerce uses the definition of 5 U.S.C § 3542b (Code of Laws of the United States (U.S.C.))[1]:

> 'The term "information security" means protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction...'

The International Organization for Standardization (ISO) defines it as follows[44]:

> 'Preservation of confidentiality, integrity and availability of information. Note: In addition, other properties, such as authenticity, accountability, non-repudiation and reliability can also be involved.'

Blakley, McDermott and Geer have a more risk management-oriented definition of information security[9]:

> '...information security is a risk management discipline, whose job is to manage the cost of information risk to the business.'

What all definitions have in common is the description of the goal: to prevent damage caused by violations of information security. Threats to this goal come in numerous kinds and forms, such as malware (viruses, worms, Trojan horses), targeted attacks of malicious hackers, system malfunctions, but also physical aspects like theft or destruction of relevant digital documents. In this work, the focus lies on information security topics related to digital information objects whose security measures take place inside a computer system – as opposed to non-digital information artefacts, or information security outside such a system.

This part of information security is called computer security and is defined in literature as:

> 'Measures and controls that ensure confidentiality, integrity, and availability of the information processed and stored by a computer.' by the Committee on National Security Systems (CNSS) [18].

A concept that seems to be omnipresent when it comes to information security in general and computer security particularly, is the CIA Triad.

2.1.1 CIA Triad

When it comes to information security, the three main objectives to achieve are confidentiality, integrity and availability (Confidentiality, Integrity, Availability (CIA)). Together they form the 'CIA Triad'.

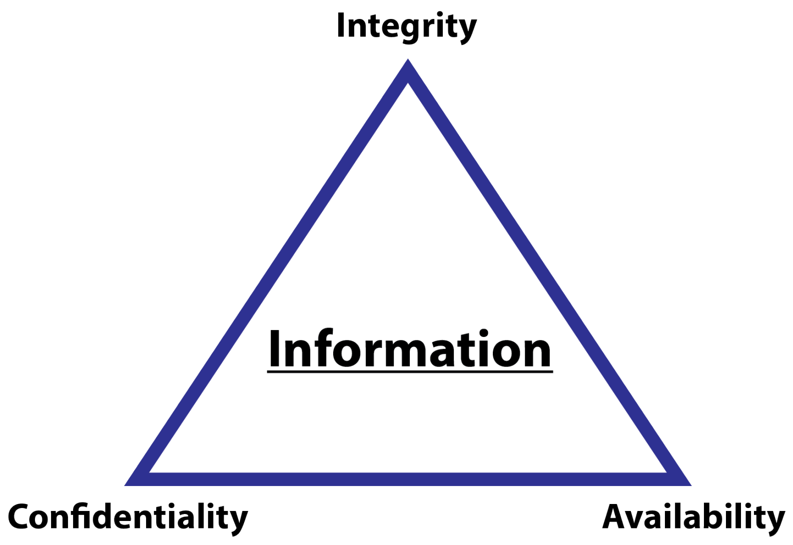

Figure 1: The CIA Triad: Confidentiality, Integrity, Availability

Confidentiality

Eckert[25] defines: ‘Confidentiality is provided, when there is no unauthorised information disclosure’. It ‘is the concealment of information or resources’, as Bishop[8] states. In other words: only people who are authorised to read information should be able to do so. Examples of violated information confidentiality could be:

- Installing a key logger on a colleague’s keyboard and thereby obtaining a password.
- A man-in-the-middle attacker who reroutes a network’s traffic in order to gain information.
- Eavesdropping on a phone call from a third party.

Further, the concept of confidentiality also has relevance when it comes to the mere existence of data. It might, for example, be of importance for a political party to keep the existence of certain documents a secret, or for an inventor to make sure that competing firms do not know about the newest inventions.

To ensure confidentiality, there are a lot of technologies in IT security. The most immediate ones are access control mechanisms, as described by Ballad[7], that ensure that an individual can only access information he/she is allowed to. Sometimes, however, access control is simply not possible. Packets that are transmitted via Wi-Fi, for example, are accessible by anyone. A way to guarantee confidentiality is cryptography, as described by Bishop[8] and later in this Section. An attacker might read the encrypted data, but cannot obtain the clear text out of them without the right key or secret.

There are two approaches to implement access control: in the first, the person can access every information, except the one that he/she is not allowed to see (this is called a ‘black list’ approach[7]). The second one would deny the person to see any information, except the one that he/she needs to work (this is called a ‘white list’ approach[7]). The black list approach has the advantage that the individual is more likely to have access to all the information needed to work effectively, thus reducing organisational overhead when asking for access rights. On the downside, however, it is more likely that a person obtains access to a confidential piece of information by accident, as the blacklist must be kept updated. With the white list approach it is vice versa, which makes it especially effective in areas of high confidentiality needs, where compromised confidentiality of information can lead to severe consequences (e.g. in military environments, or intelligence agencies).

A further method for access control is Mandatory Access Control (MAC). Pfleeger[71] defines MAC as follows:

> ‘Mandatory access control (MAC) means that access control policy decisions are made beyond the control of the individual owner of an object. A central authority determines what information is to be accessible by whom, and the

> user cannot change access rights. An example of MAC occurs in military security, where an individual data owner does not decide who has a top-secret clearance; neither can the owner change the classification of an object from top secret to secret.'

He then continues to describe Discretionary Access Control (DAC) as an access control method in contrast to MAC and outlines how they relate to each other:

> 'By contrast, discretionary access control (DAC), as its name implies, leaves a certain amount of access control to the discretion of the object's owner or to anyone else who is authorized to control the object's access. The owner can determine who should have access rights to an object and what those rights should be. Commercial environments typically use DAC to allow anyone in a designated group, and sometimes additional named individuals, to change access. For example, a corporation might establish access controls so that the accounting group can have access to personnel files. But the corporation may also allow Ana and Jose to access those files, too, in their roles as directors of the Inspector General's office. Typically, DAC access rights can change dynamically. The owner of the accounting file may add Renee and remove Walter from the list of allowed accessors, as business needs dictate. MAC and DAC can both be applied to the same object. MAC has precedence over DAC, meaning that of all those who are approved for MAC access, only those who also pass DAC will actually be allowed to access the object. For example, a file may be classified secret, meaning that only people cleared for secret access can potentially access the file. But of those millions of people granted secret access by the government, only people on project "deer park" or in the "environmental" group or at location "Fort Hamilton" are actually allowed access.'

Another important technology for ensuring information confidentiality is cryptography, as described by Bishop[8]. Even if an attacker succeeds in retrieving the relevant data, it is not possible to retrieve the relevant information without the cryptographic key. A person with malicious intent now has two technical possibilities: find a weakness in the cryptographic algorithm, or find the key. Cryptography is only as secure as the key's confidentiality and quality (it must not be guessable) and the algorithm's security, as stated by Kerkhoff[48].

Integrity

The NIST defines integrity as 'guarding against improper information modification or destruction, and includes ensuring information non-repudiation and authenticity'[75]. This means that data itself, as well as its origin, must not be manipulated in an unauthorized way. Integrity is the basis of every functioning system, as information exchange always relies on the trust in the information's credibility. Here are some examples to illustrate the concept of integrity:

- An attacker changes the last-modified date of a file. The data's integrity is broken, as it might be of importance when a file was last modified in a criminal case.
- In a company network, a Man in the Middle (MITM) intercepts emails with important account information, manipulates the information and forwards the emails to the recipient. The information's integrity is violated.
- During a file download, an error occurs and the file ends up incomplete on the hard disk. The integrity of the file is broken.

An important technology to indicate data modification is hash functions[8]. Hash functions are functions that, for a given block of data generate a hash value of fixed length. It must be computationally infeasible to find the original input data from the hash value or generate arbitrary input data, from which the function would generate the same result. By comparing hash values, it is possible to detect data modification. Of course this only works if the hash value to compare it with is sufficiently secured and cannot be modified itself. Another method for detection of integrity violations are digital signatures. 'A digital signature is a construct that authenticates both the origin and contents of a message in a manner that is provable to a disinterested third party'[8]. Through the signature of the sender, the recipient can verify the sender is indeed who he/she claims to be and that after the artifact's signing it has not been modified in any way.

A measure to preserve a file's integrity is denying access to it in the first place through access control measures[7]. However, random changes through e.g. I/O errors, are then still possible. Further, access cannot always be restricted. Network packets can be intercepted and modified by a MITM. Cryptography is a way to maintain integrity nevertheless.

Availability

Availability is defined as 'ensuring timely and reliable access to and use of information' [1]. It is imperative for a system to work when needed and as needed, otherwise it is useless. Examples of availability violations are as follows:

- Due to a power failure, a server had to reboot. During the reboot procedure, a kernel error occurred and it remained in a state error and unresponsiveness.
- A web server can only handle a maximum of n concurrent connections. After the number is exceeded, the web server is unreachable.
- An attacker finds a vulnerability in an eBanking service and manages to crash it. Users cannot reach their account data any more.

A great risk to a system's availability is posed by so called Denial of Service (DoS) attacks, in which an attacker renders the system useless. This can be done by exploiting a weakness, which brings the target to a crash, feeding it data that keeps the system busy so that it cannot handle other work, or by overwhelming a service or network with more requests than it can handle, thus rendering it unresponsive. A variant of DoS attacks are Distributed Denial of Service (DDoS) attacks, in which an attacker gains control over a lot of third party computers and facilitates this armada of 'zombie' clients to attack a target. This results in huge amounts of traffic, making the target unreachable. Various protocols can be used for different degrees of severity in DDoS attacks, as described by Rossow[76]. He outlines how the data flow is amplified by some protocols. This is called amplification attack[76]. According to a survey with over 849 companies in North America, Europe and the Asia-Pacific region, the average revenue risk of DDoS attacks amounted to USD 2.5 million, as Neustar[59] described.

Important measures to reduce the risk non-availability include redundancy in services, computers and infrastructure and secure coding measures to reduce the risk of vulnerabilities that lead to DoS. To prevent the kind of DoS attacks, where a malicious instance tries to overflow the network with requests (e.g. DDoS attacks), the target needs to cooperate with its Internet Service Provider (ISP) or other service providers positioned between the internet and the attacked network, as the huge amount of data needs to be blocked before it reaches the target's network, as Peng[69] describes.

2.1.2 Privacy

The defining of the term of privacy or the right to privacy is a difficult task, as various topics of privacy rights are also protected by other laws (e.g. laws against trespassing, stalking, bodily harm, etc.) and the concept of privacy is a subjective one and may be seen differently from culture to culture, as Onn argues [62]. He then continues to try a definition:

> 'The right to privacy is our right to keep a domain around us, which includes all those things that are part of us, such as our body, home, thoughts, feelings, secrets and identity. The right to privacy enables us to choose which parts in this domain can be accessed by others, and control the extent, manner and timing of the use of those parts we choose to disclose.'

Since May 2018, privacy is handled EU-wide by the EU General Data Protection Regulation (EUGDPR), as described in Section 2.1.3.

The common criteria specification breaks down the topic of privacy in four functional requirements [19] as illustrated in Figure 2:

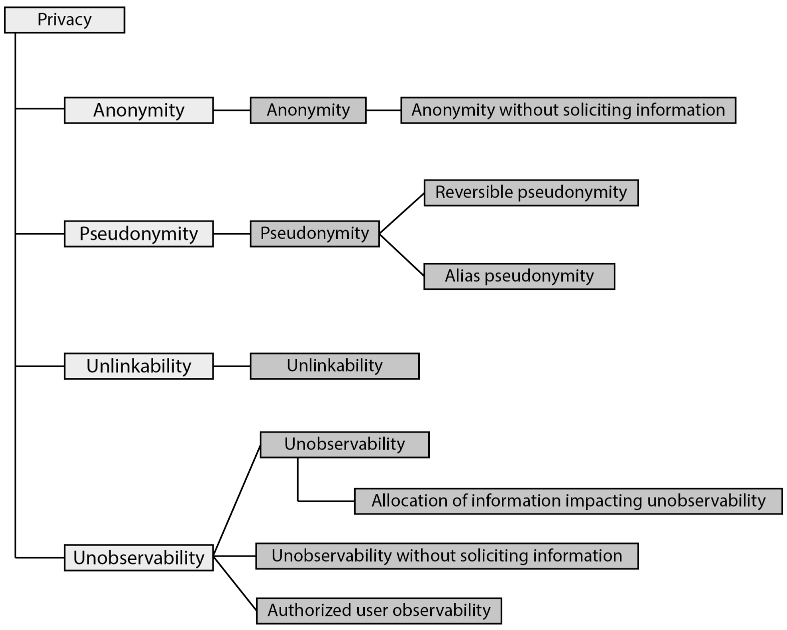
Privacy
Anonymity
Anonymity
Anonymity without soliciting information
Pseudonymity
Pseudonymity
Reversible pseudonymity
Alias pseudonymity
Unlinkability
Unlinkability
Unobservability
Unobservability
Allocation of information impacting unobservability
Unobservability without soliciting information
Authorized user observability

Figure 2: Common Criteria Class Decomposition; rebuilt after Figure 19.4 of Stallings[87]

Anonymity

The requirement for anonymity protects a user's identity, so that the user may use a service without other subjects being able to establish to user's true identity. Anonymity without soliciting information means that anonymity is established without the system even asking for the individual's identity[19].

Pseudonymity

Pseudonymity means that other users are not able to directly determine a user's identity. The user is identified by a pseudonym, e.g. an alias (alias pseudonymity) or another constructed pseudonym, which is based on certain construction rules. It is only possible to truly identify a user with a mapping of true identity to pseudonym. This way certain privacy is provided to the user, without losing the ability to still hold the individual accountable for actions taken[19].

Unlinkability

Unlinkability is given if different operations and uses of a service from the same user cannot be linked to each other. If user A performs two actions, X and Y, user B must not be able to determine that both actions X and Y are performed by the same user[19].

Unobservability

A system fulfils the requirements of unobservability when it is impossible for a user to determine if an operation is performed or a service used. The allocation of information that impacts unobservability requires that privacy related information must not be concentrated in a system or subsystem, as these information might be relevant to the topic of unobservability. Unobservability without soliciting information means that the system must not obtain information that might be used to compromise unobservability. Authorised user observability allows authorised users to observe a service's usage[19].

Of course, the topic of privacy with all its implications in the fields of law, national security, ethics, philosophy, etc. goes far beyond the given definitions and standards. A popular public debate, for example, is the one between national security and an individual's privacy. These discussions go beyond the scope of this work.

2.1.3 Legal Situation

As legal situations vary from country to country, this section focuses on Austrian and European law.

The Data Security Regulation[94] regulates internal security requirements for providers and data security requirements when transmitting mission-critical traffic- and location data to authorities.

Several mentions of data security or data protection, e.g. in Section 12 of the Telecommunication Law 2003[13] or §31a General Social Security Law[15] refer to the EU General Data Protection Regulation (EUGDPR)[67] and the data protection adaption law [12], which prepared the Austrian legal system for the EUGDPR and invalidated the previous data protection law[14].

EU General Data Protection Regulation

In April 2016 a EU-wide new data protection regulation was published in the EU Official Journal and enters into force [67]. Therein written laws are directly binding and applicable in all EU member states and enforced as of 25 May 2018.

Articles 2 and 3 state that the regulation applies to the processing of personal data by a controller (an organisation which collects data) or a processor (an organisation which just processes the data on behalf of a third party), if the data subject is based in the EU, regardless of whether the controller or processor is in- or outside the European Union. According to Article 4, *personal data* is any information relating to an identified or identifiable natural person.

Article 9 mentions special categories of personal data, including

> '...racial or ethnic origin, political opinions, religious or philosophical beliefs, or trade union membership, and the processing of genetic data, biometric

> data for the purpose of uniquely identifying a natural person, data concerning health or data concerning a natural person's sex life or sexual orientation. . . '

Processing of such data is prohibited, except in cases defined in §2, Article 9.

The data subject has the right to know if, which, and for which purpose a controller processes the subject's personal data (Article 15). Articles 16 and 17 state the right to rectification and erasure of ones data. Every data subject also has the right to data portability, meaning that he/she is allowed to obtain the personal data collected by a controller in machine-readable format and have the right to transmit those data to another controller (Article 20).

Article 25 demands data protection by design. The controller must implement data-protection measures and principles to integrate the necessary safeguards and meet the regulation's requirements. It further states that the controller must implement data protection by default:

> 'The controller shall implement appropriate technical and organisational measures for ensuring that, by default, only personal data which are necessary for each specific purpose of the processing are processed. That obligation applies to the amount of personal data collected, the extent of their processing, the period of their storage and their accessibility. In particular, such measures shall ensure that by default personal data are not made accessible without the individual's intervention to an indefinite number of natural persons.'

Article 51 demands the establishment of an independent supervisory authority, which, among other tasks, monitors the application of the regulation, promotes public awareness, advises the government and handle complaints by data subjects. Every member state's head of the supervisory authority is also member in the European Data Protection Board, a body of the European Union.

As stated in Article 37, a designated data protection officer must be appointed by the controller or processor, if:

- the processing is carried out by a public authority, or
- the controller or the processor executes operations that require regular and systematic monitoring of data subjects on a large scale, or
- the controller or the processor process data that belong to a special category, as discussed in Article 9 or data relating to criminal convictions and offences.

The data protection officer has several tasks, including: advising and informing the controller or processor of personal data, and its employees about their obligations, monitoring compliance with the EUGDPR and cooperating with the supervisory authority.

In case of a data breach, the controller must notify the responsible supervisory authority within 72 hours (Article 33). If the data breach is likely to result in high risk to the rights and freedoms of natural persons, the controller must inform the data subject without undue delay and communicate in clear and plain language the nature of the personal data breach.

2.1.4 Protection Needs & Risk Management

In a system, different objects have different needs for measures of protection of their data security. To give every object the same level of protection would be either insufficient (if the level is too low) or economically infeasible (if the level is too high). The protection needs of components or processes states, how much protection it needs, depending on how important the component or process is for smooth business operation. This helps to assess its importance to fulfill the given security goals [11]. The German Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik (BSI)) proposes three categories of protection need [11], depending on severity of possible damage:

- normal – the damage's impact is limited and manageable.
- high – the damage's impact can be substantial.
- very high – the damage's impact can be existentially threatening and disastrous.

To help define an object's protection need category, the BSI[11] presents the following scenarios and gives examples in Tables 1, 2 and 3 that indicate which kind and severity of damage could belong to which category:

- Infringement of laws, regulation or contracts,
- Impact on informational self-determination (meaning how it affects the authority of an individual to decide about the communication of personal information, regarding him or her, to others),
- Impact on personal integrity,
- Impact on fulfilment of tasks,
- Negative image inside or outside the organisation and
- Financial impact.

Table 1: Category ‘normal’ protection need examples [11].

Scenario	Example
1. Infringement of laws, regulation or contracts	• Violations against regulations and laws with minor consequences • Minor contract violations yielding minimal penalties
2. Impact on informational self-determination	• Processing of personal data, which can influence social or economic position
3. Impact on personal integrity	• Impact does not seem possible
4. Impact on fulfilment of tasks	• Impact is assessed as tolerable by affected persons • Maximum tolerable down time is greater than 24 hours
5. Negative image inside or outside the organisation	• A small or only intern impact on reputation or trust
6. Financial impact	• The organisation’s financial impact stays tolerable

Table 2: Category ‘high’ protection need examples [11].

Scenario	**Example**
1. Infringement of laws, regulation or contracts	• Violations against regulations and laws with considerable consequences • Contract violations high penalties
2. Impact on informational self-determination	• Processing of personal data, which can influence social or economic position considerable
3. Impact on personal integrity	• Physical injury to an individual cannot be absolutely ruled out
4. Impact on fulfilment of tasks	• Impact is assessed as intolerable by single individuals • Maximum tolerable down time is between 1 and 24 hours
5. Negative image inside or outside the organisation	• A wide impact on reputation or trust can be expected
6. Financial impact	• The organisation’s financial impact is considered remarkable, but not yet existentially threatening

Table 3: Category 'very high' protection need examples [11].

Scenario	Example
1. Infringement of laws, regulation or contracts	• Fundamental violations against regulations and laws • Contract violations, which penalties are ruinous
2. Impact on informational self-determination	• Processing of personal data, which poses a danger for life and limb or personal freedom
3. Impact on personal integrity	• Serious impact on personal integrity is possible • Danger for life and limb
4. Impact on fulfilment of tasks	• Impact is assessed as intolerable by all individuals • Maximum tolerable down time is lower than one hour
5. Negative image inside or outside the organisation	• A country-wide impact on reputation or trust, eventually even of existentially threatening nature, can be expected
6. Financial impact	• The organisation's financial impact is existentially threatening

The above tables can serve as a guideline in assessing a component's protection needs level, however, each organisation needs to find its own applicable definitions.

When evaluating an application's level of protection needs, *dependencies* needs to be considered as well. If, for example, application B has high protection need and depends on input for one application A, the protection need from application A might increase.

A system's protection needs is always evaluated towards a value, e.g. the three main features of information security: confidentiality, integrity and availability, as described in

Section 2.1.1. Moreover, when establishing an IT system's protection need, including subsystems with their own protection need, there are a few principles that need consideration. These principles take effect when it comes to a system containing different components interacting with each other.

The *maximum principle* states that an IT system's protection needs is at least as high as the highest protection level of an application on the IT system. Figure 3 shows an example of the maximum principle. While subcomponent 1, 3 and 4 have normal protection need, subcomponent 2 has high protection need. Following the maximum principle, this leads to high protection needs for component X.

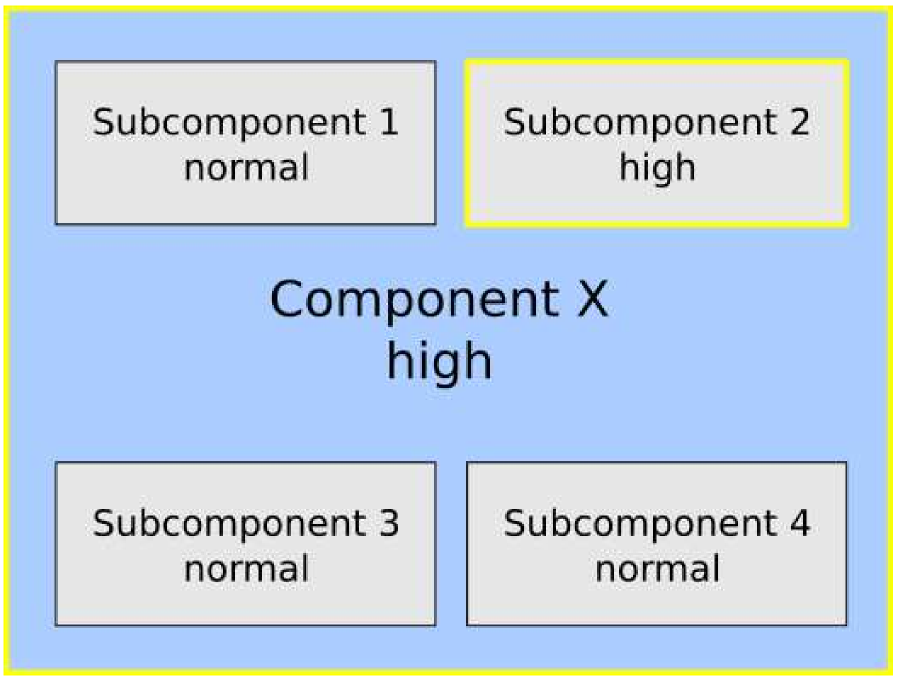

Figure 3: Example of the maximum principle.

Due to the *cumulative effect* several smaller damages can cumulate to a bigger damage for the whole system, thus yielding in a higher protection needs. As shown in the example of Figure 4, subcomponents 1–4 have high protection needs. Through the cumulative effect, it is possible that component X ends up with very high protection needs, as the subcomponents' protection needs cumulate.

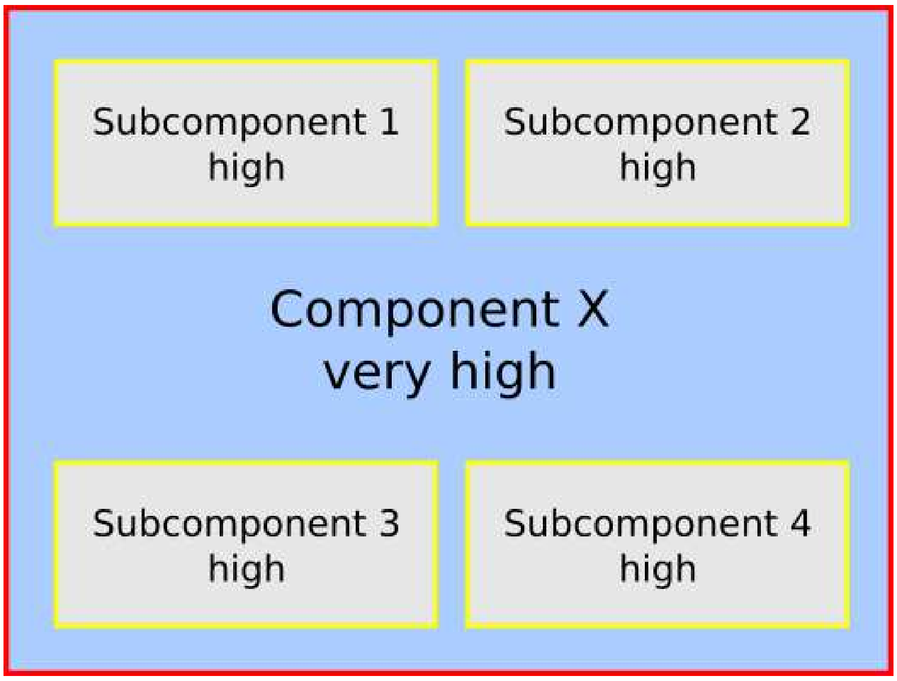

Figure 4: Example of the cumulation effect.

The *distribution effect* states the possibility that an application's high protection needs are not passed on to the containing IT system, if it contains only unessential parts of the whole system, or if the application is designed redundantly. Figure 5 shows an example where subcomponents 1.1–1.4 are redundancies of a subcomponent 1. This redundancy leads to a lower protection need for the logical cluster X.

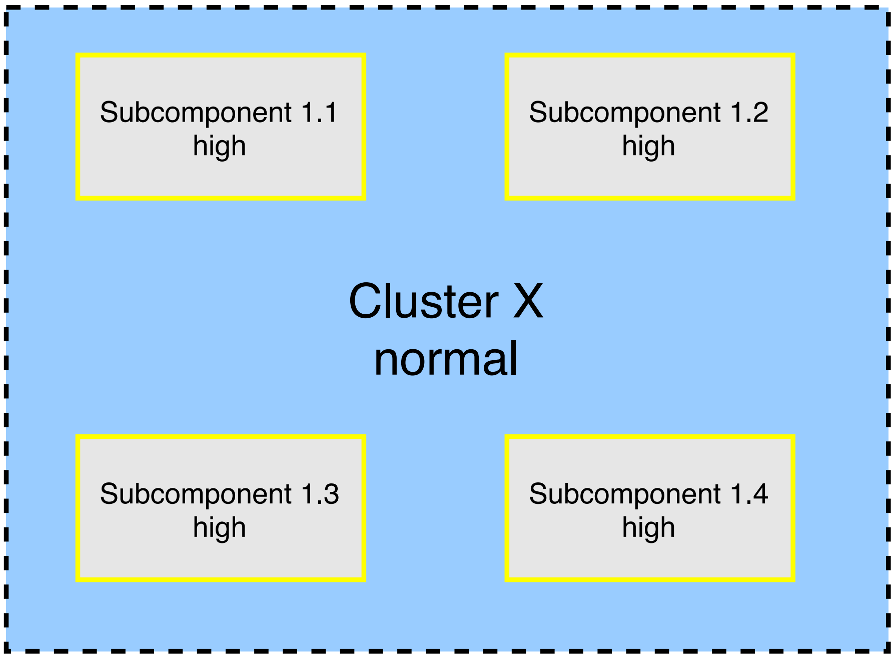

Figure 5: Example of the distribution effect.

Risk Management

The assessment of damage to a system (e.g. by defining the level of protection needs), or parts thereof, is an important factor in applied risk management.

The International Organization for Standardization (ISO) defines risk as the impact of uncertainty on goals and risk management as coordinated activities to guide and control an organisation in regard to risks [64]. The Austrian Standards Institute denotes risks as the impact of uncertainty on goals, tasks and requirements and risk management as processes and behaviour designed to control an organisation in regard to risks[65].

Figure 6 shows the steps of a risk management process as suggested by ISO 31000 [64], which are described in detail below.

- *Communication and Consultation* should happen internally and externally during every step of the process, so every person responsible and every stakeholders has the basis for making proper decisions and understands the reason for various measures.
- *Establishing the Context* helps to formulate the organisation's goals, define internal and external influencing factors, set the scope and criteria, for the following risk management process.
- *Risk Identification*, the first step of risk assessment, is necessary to identify possible sources of risks, affected areas, events and changes, as well as their causes and potential effects.

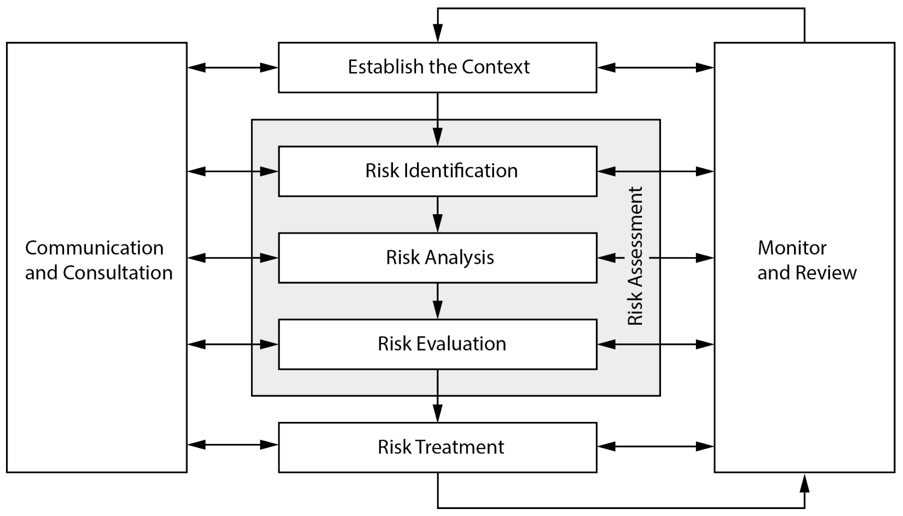

Figure 6: Risk management process as suggested by ISO 31000 [64].

- During *Risk Analysis*, a proper understanding of the risks at hand is established. This includes information like risk probability, type and extent of possible impact, complexity and connectivity, and time factors.
- *Risk Evaluation* is necessary to decide the necessity and priority of taking measures to mitigate identified risks, based on the previous risk analysis. There are different ways to value a risk, e.g. in terms of low, medium, high, or with a product of probability * damage.
- *Risk Treatment* is the selection and execution of measures to mitigate risks.
- To *Monitor and Review* during the whole risk management process is essential to continuously adapt and improve the process and learn about the effectiveness of measures taken.

2.1.5 Software Security

Every piece of software has the potential to contain security vulnerabilities. Alhazmi et al.[3] show a rise in vulnerabilities, using the example of Microsoft Windows and Redhat Linux. This results in an increased need for software security. According to McGraw[55], the rise is explained by three factors:

- Connectivity: the growing connectivity of computers through the Internet has increased both the number of attack vectors and the ease with which an attack can be made.
- Extensibility: through a rising number of extensible applications, vulnerabilities in such extensions can easily slip in.

- Complexity: the growing size and complexity of software increases the possibility of bugs and vulnerabilities. Given the example of the Linux kernel, version 0.01 from 1991 had 5,929 lines of code, whilst version 5.2.2 from 2019 already has 18,597,392 lines of code (only lines of *C* code counted with script from Listing A.1.

McGraw[55] further identifies three main pillars of software security to help solve the problem:

- Applied Risk Management: thorough management of the risks at hand during the whole software development life cycle is necessary to constantly identify, assess and mitigate risks during the whole development process.
- Software Security Touchpoints: awareness that security cannot be added later on, but must be worked into the software during the whole software development process is an essential approach. Planning on which security measure (audits, security testing, external reviews, ...) should be applied in which step of the development life cycle helps in establishing a wholistically secure development process.
- Knowledge: the basis for secure software is that everyone involved (developers, architects, testers, ...) has sufficient knowledge to plan, implement and, if necessary, adapt the security measures. Education/training is essential for achieving this goal.

Finding bugs early in the software development process is cheaper than finding them later, as McConnell depicts[54]. Table 4 shows how the relative cost of fixing bugs changes with the phases of the software development life cycle.

Table 4: Relative cost of fixing defects based on when they are introduced and detected (Requirements, Architecture, Construction, System Test or Post-Release), by McConnell[54]

	Time Detected				
Time Introduced	Requ.	Arch.	Constr.	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture	–	1	10	15	25-100
Construction	–	–	1	10	10-25

The conclusion is, as discussed by McGraw[55] that security must be a part of the whole software development life cycle to identify and mitigate security vulnerabilities as early in the process as possible. Geer[33] analysed a survey about secure development life cycle models, of which there are various. As the Microsoft Security Development Lifecycle (SDL) enjoyed the most awareness, it will be described in more detail. Microsoft developed the SDL, a process to follow along, and to be interwoven with the regular software development process to increase software security. The steps, as described by Howard and Lipner [41] are:

1. Education and Awareness: education and security awareness of everyone involved is the basis for the process to succeed.

2. Project Inception: at the beginning of the project, some fundamental tasks are to be done, like assigning people responsible for security, defining communication and responsibilities between developers and security team and decide which kinds of security bugs shall be reported and which not.

3. Define and Follow Design Best Practices: there exist several best practices and principles for secure design that can help to lay the foundation for secure development in the design phase. Further, a thorough analysis of the product's attack surface helps to understand possible threats and assists in reducing the given attack surface.

4. Product Risk Assessment: in this phase, all the project portions are assessed as to the risks given and which security measures are necessary to reduce or mitigate that risks. A privacy assessment of the data used helps to identify the project's parts that process delicate data and, if breached, could cause great damage.

5. Risk Analysis: threat modelling helps to identify possible threats to the product's security. It gives good insight into various points of attack, helps to uncover threats to the application and assists in evaluating the severity of possible damage.

6. Creating Security Documents, Tools, and Best Practices for Customers: security documentation and best practices help a user understand the implications of his/her actions and configuration changes. This makes it easier to use the product in a secure way. Security tools can help to assess the current configuration and assist in tuning the configuration.

7. Secure Coding Policies: there are many secure coding guidelines and best practices. It is equally important to use them as it is to understand them properly. Source code analysis tools can help identify flaws, but do not replace developer skills.

8. Secure Testing Policies: security testing is necessary to find security bugs in a product. Non-security experts can make use of fuzzing tools to test for different input values, however, critical applications should always be tested by an experienced penetration testing team.

9. The Security Push: a security push is a time-boxed phase in which the whole team can focus on security and filing security issues. Keeping the threat-models and attack surface documentation up-to-date helps in shortening the time needed.

10. The Final Security Review: Before the product ships, during the final security review the application's readiness to release is checked from a security and privacy viewpoint.

11. Security Response Planning: A security response team must be prepared to respond to security issues, when they arise. Proper planning of responses to discovered security issues is the basis for efficient elimination of security vulnerabilities

12. Product Release: if the SDL has been followed properly, the product can be released.

13. Security Response Execution: no product is perfect and security issues are likely to arise. They must be handled according to the security response plan already prepared. Once a bug is found, the product must also be checked for similar or related vulnerabilities.

Following this security development life cycle and integrating it into the software development project can help to detect security issues as soon as possible and prevent creating them in the first place.

2.2 Software Testing

Myers has the following two definitions of software testing [58]:

> ‘Testing is the process of executing a program with the intent of finding errors.’

> ‘Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and, conversely that it does not do anything unintended.’

In essence, software testing is a way to try and improve a software product’s quality by reducing the number of bugs, eliminating other unwanted behaviour and ensuring that the product does what it is intended to do.

In software testing, three main tasks need to be performed for each test case[58]:

1. Define test data to execute the test case accordingly.

2. Declare the output that is expected by the SUT.

3. Evaluate the result by comparing the actual output by the SUT to the expected output.

2.2.1 Requirements

Identifying unwanted behaviour can sometimes be easy, even obvious (e.g. when executing a command and a big, red error message presents itself) and sometimes, it can be much more subtle (e.g. the thirteenth decimal place in a division is off by one, but only under specific circumstances). As the latter are much harder to detect, the likelihood of only finding them in a later project phase, after release or not even at all, is much higher, thus rendering the impact much more expensive, as shown in table 4.

In order to make sure such non-obvious errors are found as well, it is necessary to define what a specific function should, and what it should not, do. The more clearly and in more detail the requirements are defined, the easier it is to create test cases that cover these requirements in detail and find related bugs. Requirements are the basis for software testing. Therefore, it is essential to dedicate enough time to requirement analysis in the conception phase of a software development project.

Requirements can be roughly classified into two types: functional and nonfunctional requirements. In the Software Engineering Body of Knowledge [10], these are defined as:

> 'Functional requirements describe the functions that the software is to execute; for example, for- matting some text or modulating a signal. They are sometimes known as capabilities or features. A functional requirement can also be described as one for which a finite set of test steps can be written to validate its behavior. Nonfunctional requirements are the ones that act to constrain the solution. Nonfunctional requirements are sometimes known as constraints or quality requirements. They can be further classified according to whether they are performance requirements, maintainability requirements, safety requirements, reliability requirements, security requirements, interoperability requirements or one of many other types of software requirements'

The discipline that includes the handling of requirements during the whole life cycle of a system is called requirements engineering. Dick[21] defines requirements engineering as:

> 'the subset of systems engineering concerned with discovering, developing, tracing, analyzing, qualifying, communicating and managing requirements that define the system at successive levels of abstraction.'

This definition contains several activities that need explanation[21]:

Discovering:	The discovering of requirements includes the elicitation and collection of requirements from stakeholders.
Developing:	Developing the requirements means adding details as necessary for various phases in the development process, or updating them.

Tracing: The tracing activity is tracing the requirements to various other artefacts, like test cases that verify the fulfilment of the requirement or other requirements. This trace makes it easier to directly verify and validate the results against the requirements. To be able to do that, tracing also includes uniquely identifying each requirement by assigning a unique identifier, e.g. a number.

Analysing: Analysing includes the analysis of requirements for economical and technical feasibility and analysis if the stated requirement covers the real need of the stakeholder.

Qualifying: About qualifying Dick[21] writes: 'This refers to all kinds of testing activity, covering testing of the design and solution, including unit, component, integration, system, acceptance testing. There is considerable disagreement over the meaning of the terms "verification" and "validation." The term "qualifying" is preferred, because it is about ensuring that the solution has the required "qualities." In so much as the terms are used in this book, to validate requirements is to check a formal expression of requirements against informal needs as understood in the minds of stakeholders, and to verify requirements is to check their internal consistency within layers and between layers of abstraction.'

Communication: The Communication between suppliers, customers, users, developers and whoever is stakeholder in the project, is essential to gain a common understanding on what has to be done.

Especially in projects with numerous requirements, a structured approach can help to manage requirements effectively.

2.2.2 Verification vs. Validation

The Institute of Electrical and Electronics Engineers (IEEE)[42] defines verification as:

> 'The process of evaluating a system of component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase' and as 'formal proof of program correctness'.

This means that the verification of a product questions whether it is built the right way and if the implementation accords to the specification. If a function should sum up two summands and it would return 5 for the input parameters of 2 and 2, this would be a verification failure.

Validation is defined as follows[42]:

> 'The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.'

This contrasts with the definition of verification, as verification only checks for satisfaction of conditions imposed at the start of a given phase to have a formal proof of correctness, and validation is the check against the initial requirements and needs of the customer.

The validation of a product asks the question, if the right product has been built. It is to check if the specification and the result correspond to the customer's needs and requirements. Validation would fail, for example, if the customer would express the need for above discussed sum-function, and would instead get a multiplication function. Although it works perfectly fine and for the parameters of 2 and 2 even renders the same result, it is not what the customer asked for and is, therefore, invalid.

Having in mind the differences between verification and validation is important in software testing. The developer product needs to be not only functioning correctly, but it also needs to be the correct product. Valid software products are the result of proper and frequent communication with the customer. A software product can be perfectly functioning, but useless to the customer. On the other side, it can be exactly what the customer wanted, but behave erroneous.

A good software product needs to pass verification and validation, both of which can be accomplished by establishing measure of software quality assurance in every project phase.

2.2.3 Typical Testing Process

Spillner[86] suggests the structured software testing process shown in Figure 7 and described as follows:

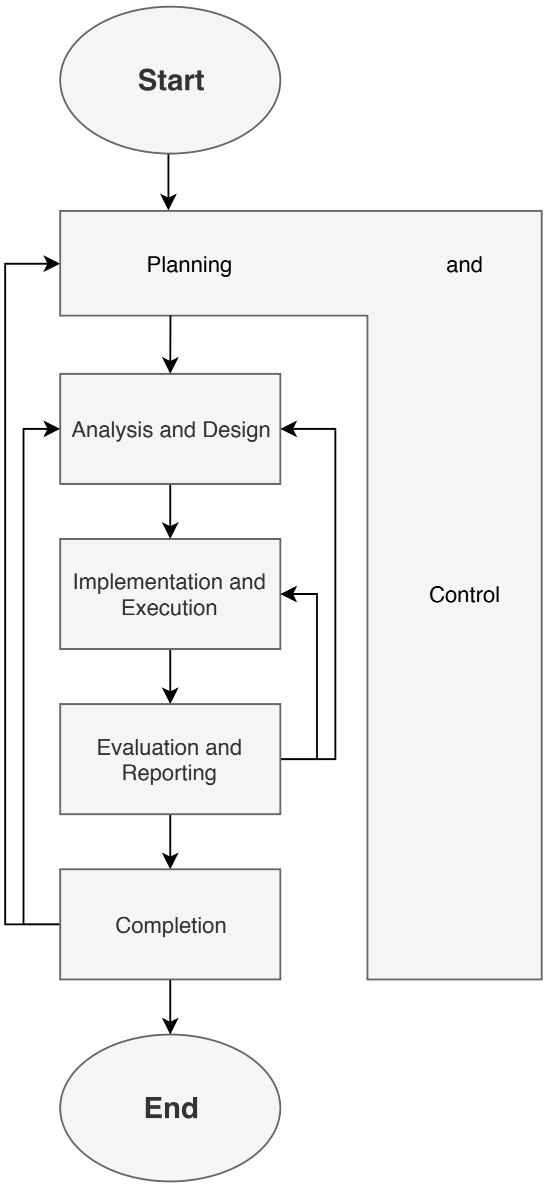

Figure 7: Different steps of a testing process, rebuilt after Spillner[86]

1. The process starts in the test planning phase, where necessary preparations for a successful structured testing process are made, including allocation of needed personnel and other resources, facilities and tools and matters of time planning. Further organizational questions, like assignment of roles (who does what in the process?), description and distinction of the system under test, determination of the test strategy, prioritizing of tests and defining exit criteria are, together with all other information from the planning phase, written down in the test plan. Measures of regular test control are established to see if test planning and execution match.

2. In the test analysis and design phase, the test basis is verified to see if requirement specifications are accurate enough, or if the chosen test strategy may be unsuited for the test object. Further, logical test cases are designed for implementation in the following phase.

3. The implementation and execution phase comprises the implementation of the previously defined logical test cases and execution thereof. After that, the execution of test cases, in the previously defined prioritisation, yields the desired test results. It is of essence to document the defined test cases, the execution and the results in the test report, in order to make findings replicable and document the test coverage. After a bug is fixed by development, the failed test case must be re-executed for verification.

4. In the test evaluation and test report phase, test execution is checked for meeting the defined exit criteria. If those criteria are met, the process continues to the completion phase. Otherwise, another test iteration may be necessary. It is also possible to detect that the exit criteria are unreasonably high and its fulfilment is infeasible. In practice, the test process is often closed by time and budget constraints. Before continuing to complete the test process, the report should be finished, as it is easier to recall important facts and document as soon as possible and as a deliverable, the report is necessary to make findings replicable and document the test coverage.

5. The completion phase is important to gather and document lessons learned, evaluate the whole test process and distribute knowledge gained to whom it may concern.

2.2.4 Test Cases

A test case is a plan to execute a program, or more generally test a system, to check for specific requirements. There is defined input data and the SUT should deliver a certain output. It is the building stone of a test suite or test concept and together with other test cases, it covers some elements of a SUT.

More formally defined, a test case is a set of $\{P, I, A, R\}$, where

1. P is a set of preconditions that need to be fulfilled in order to execute this test case properly. This might be that a system is up and running, a certain database state or a specific application state.

2. I represents a set of input parameters which make a difference in the execution that lead to the results.

3. A stands for the actions conducted during the test case execution, given the input parameters I. This might be, for example, a function call, file system operations or network transmissions.

4. R is the set of expected results, e.g. a certain value or a data base state.

A software test comprises a number of test cases, which can be a large amount for complex systems. Techniques to reduce the number of test cases, will be discussed in Chapter 3.

2.2.5 Test Coverage

The more parts of a SUT that are tested, and the greater the coverage, the more is verified to work as expected. When testing a system, the ideal case would be to cover the whole system, as errors can occur in every part. This, however, can be very expensive for large or complex systems. Test coverage gives an indicator of how much of a system is covered by test cases. One approach to analyse test coverage in software testing is the consideration of the SUT's control-flow. Miller[57] suggested a logical tree to visualise the control-flow which can be used as basis for different kinds of test coverage. A logical tree is a directed graph without loops, in which every logical decision in the control flow corresponds to a branch.

The different types of test coverage render various grades of detail in which a piece of software is tested. Singh[83] identifies the following control-flow oriented testing criteria:

Statement Coverage

Statement coverage has the goal to execute every statement at least once for 100% statement coverage. Consider the exemplary program in Listing 2.1 to determine if a triangle with three given sides is valid and its control flow graph in Figure 8.

Listing 2.1: Statement Coverage Example: 'Validate triangle'

```
void validateTriangle(int a, int b, int c)
{
    if((a + b > c) && (a + c > b) && (b + c > a))
    {
        printf("Triangle␣is␣valid.");
    }
    else
    {
        printf("Triangle␣is␣invalid.");
    }
}
```

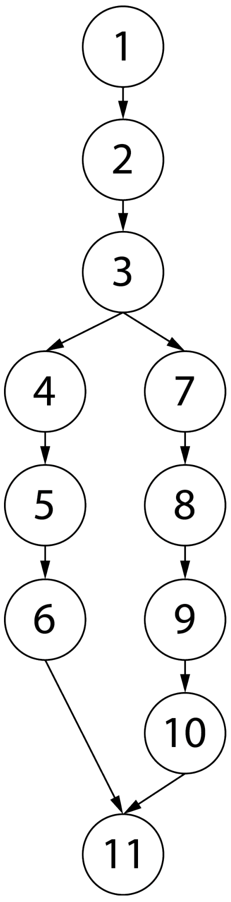

Figure 8: Control flow diagram for 'Validate triangle' program in Listing 2.1

To obtain full statement coverage, two test cases are needed, one which leads to execution of the if-branch and one for the other branch. Exemplary values could be $(a, b, c) \rightarrow (4, 4, 5), (1, 1, 3)$.

While this type of test coverage leads to a comprehensive testing of the software, it might not be sufficient. If, for example, the last two parts of the logic expression in the if-statement would be missing, the above discussed input parameters would still lead to 100% test coverage, although the program would output false results.

Branch Coverage

Branch Coverage is the covering of all branches in a program's control flow. This subsumes statement coverage, however, full statement coverage does not imply full branch coverage. The program in Listing 2.2 and its control flow graph in Figure 9 illustrates that.

Listing 2.2: Branch Coverage Example: 'Check if Alice'

```
int is_user_alice(char* name)
```

```
{
    int result = 0;
    if(strcmp(name, "alice") == 0)
    {
        result = 1;
    }
    return result;
}
```

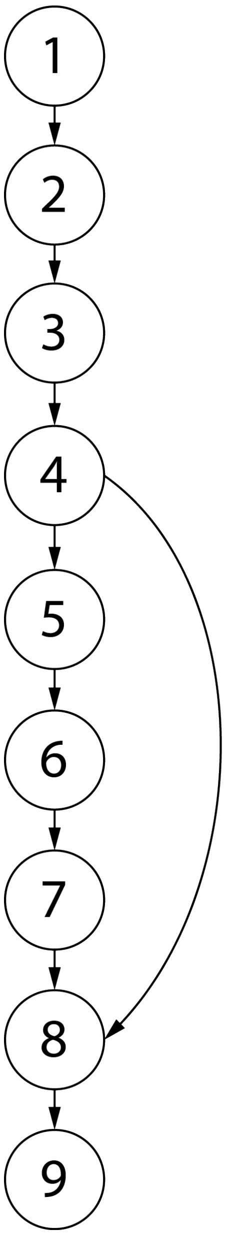

Figure 9: Control flow diagram for ‘Check if Alice’ program in Listing 2.2

While for statement coverage it is sufficient if the if-statement renders true (the given name is ‘alice’), there is a second branch the program can take, which is when the if-statement renders false. This second branch is covered by branch coverage. For statement coverage, one test case with the following input value would suffice: $name = `alice'$. Branch coverage needs two test cases: $name \rightarrow \{`alice', `othername'\}$.

Condition Coverage

Condition coverage tests every single condition at least once. This includes branch coverage, the difference is however that condition coverage not only tests for different true/false values of an overall condition, but looks at every sub-condition of the expression. This shall be illustrated using the previously listed 'Validate triangle' program in Listing 2.1 and its control flow graph in Figure 2.2.

The three sub-conditions in line 3 need two test cases each: one which renders true and one which renders false. Additionally, all combinations also need to be considered, which makes the total amount $2^3 = 8$ test cases to get full condition coverage in such a simple program:

Table 5: Needed test cases for condition coverage of the 'Validate triangle' in Listing 2.1

Nr.	**(a + b > c)**	**(a + c > b)**	**(b + c > a)**	**Values of a, b, c**
1	F	F	F	0, 0, 0
2	F	F	T	0, 1, 1
3	F	T	F	1, 0, 1
4	F	T	T	0, 0, 1
5	T	F	F	1, 1, 0
6	T	F	T	0, 1, 0
7	T	T	F	1, 0, 0
8	T	T	T	1, 1, 1

Path Coverage

For full path coverage, a test needs to execute every possible execution path a program can take. This includes every decision fork and every possible loop iteration and might therefore result in large amounts of test cases, thus rendering the test very expensive. In some cases (e.g. endless loop), it might be impossible to perform such a test. Path coverage subsumes above discussed coverage types.

2.2.6 Testing Levels

The IEEE Computer Society mentions three different testing levels [10], which differ in the target of the test. They are best visualised using the V-model as described by Alpar[4].

Unit Testing

A unit test's purpose is to test a single module or unit. The unit's size can vary. It might be a single function, a part of a function or a whole class in a program developed in an object-oriented manner.

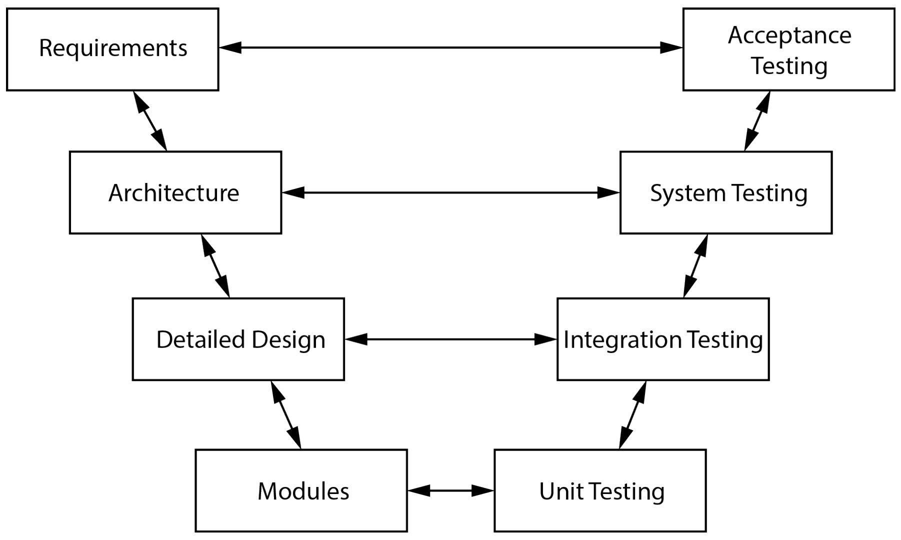

Figure 10: The V-Model as presented by Alpar[4]

Integration Testing

Integration testing is the process of testing different unit tested modules to verify their functionality as they work together. It's purpose is also to make sure that changes in one module do not affect other modules in a negative way.

System Testing

To test the whole system as such, a system test is utilised. Some defects might only be detected on this level, e.g. external interfaces to other applications, and non-functional requirements like performance, reliability and security, might also require views on the whole system.

Acceptance Testing

Acceptance testing is testing a system against previously defined acceptance criteria. This test type is typically used to determine if a product satisfies the customer's requirements. This corresponds to validation, as described in Section 2.2.2.

2.2.7 Test Types

Another way of categorising software tests is through their objective. The IEEE Computer Society mentions different test types [10] targeting various functional and non-functional properties of the system under test.

Acceptance Testing

Acceptance Testing has already been described in Section 2.2.6.

Installation Testing

After completion of a system or program, the installation on the target software- and hardware environment can be tested with an installation test. Faults and issues in the installation procedure can be detected with this test.

Alpha and Beta Testing

To get valuable feedback on the developed software, the software can be released to a smaller group of potential customers, the alpha testers and afterwards to a larger, representative group of beta testers. Upon using the product, they report issues and opinions back to the development team.

Reliability Achievement and Evaluation

When testing a system, it can be said to become more reliable, as the fact that the tested case yields no errors can be relied upon. This achievement of reliability can be supported by statistical measures using randomly generated test cases. A system's reliability can be evaluated using reliability growth models, which, given its known failures over time, provide a reliability measure.

Regression Testing

Regression testing is testing that a change in a system's component does not have unwanted side effects, and that the whole system continues to function properly. This is usually achieved by re-executing test suites that executed positively before the change was made.

Performance Testing

Performance testing checks if the SUT meets the defined performance criteria in all aspects (e.g. speed of execution, database performance, network performance, etc.).

Security Testing

The objective of security testing is to make sure that the SUT is not vulnerable to attacks and that sufficient security measures have been established to provide a specified level of security.

Stress Testing

Stress testing is the process of verifying that under heavy workload, the software or system still behaves as expected and stays stable. This also tests eventual counter-measures against the kind of DoS attacks that try to render a service useless by flooding it with input.

Back-to-Back Testing

Back-to-Back testing has the goal to compare two different variants of a piece of software. Both are executed with the same input, and outputs are then compared. This detects errors, unwanted deviations and regressions.

Recovery Testing

The objective of recovery testing is to analyse a system's capability of recovering from disasters (e.g. crash, power loss, etc.). This is especially relevant in environments where fault tolerance is of essence.

Interface Testing

In a system with interfaces between several components or interfaces to external components, interface testing is the process of checking an interface against a given interface specification and find possible deviations. One way of conducting such tests would be the creation of Application Programming Interface (API) calls.

Configuration Testing

Configuration testing verifies the functioning of software in different, previously defined configurations (e.g. different kind of users or varying environments).

Usability and Human Computer Interaction Testing

The goal of usability and human computer interaction testing is to analyse how easy (or difficult) it is for a user to understand and use a piece of software.

2.2.8 Black Box vs. White Box Testing

Patton[68] metaphorically describes *Black Box Testing* as testing with blinders on. It means testing the software without knowledge of its inside components, code or any other internal insights, thus looking at the SUT as a black box. In this type of testing, data is put in the test object, yielding output data. This data-driven approach is based on a specification of the data the test object should return for a given set of input data.

As an example for black box testing, consider the following function in Listing 2.3 and its specification to be tested:

Listing 2.3: Example of black box testing: function isEven

```
int isEven(int number);
```

'The function "isEven" takes an integer as input and returns an integer indicating whether the given integer is even (return 1) or not (return 0). Zero is considered to be an even number.'

In order to test the function, test cases need to be specified to compare the given input with the given output. Table 6 gives an example thereof:

Table 6: Exemplary test cases to black box test function ‘isEven’ in Listing 2.3

Nr.	Input	Expected Output
1	1	0
2	-1	0
3	0	1
4	1000	1
5	2	1

The test cases are specified and then executed without knowledge of the functions insides.

From the definition of black box testing the following pros of black box testing emerge:

- Effective comparison of actual and specified behaviour, as deviations from the required behaviour are directly visible and the tester is not biased by implementation details when implementing and executing the test cases[68].
- The functionality is tested independent of the implementation, as the tester does not know about the implementation itself, but tests from a user’s perspective. This spares the tester effort for implementation analysis.

The cons of black box testing are:

- Often impossible to find the actual location of a detected error, as no knowledge of the system’s internals is given.
- Requires detailed specification of SUT for the tester to understand how/what to test (e.g. API specifications), as the implementation can’t be looked up. This requires intensive preparation, which results in higher testing cost.
- No guarantee for sufficient code coverage, as the tester does not know which execution paths can be covered (the system is a black box).

In contrast to black box testing, where the tester has no knowledge of the test object’s internals, white box testing is defined by the IEEE Computer Society as ‘tests that are based on information about how the software has been designed or coded’ [10]. This normally includes access to the source code itself, thus making a detailed review and understanding of the SUT possible.

Listing 2.4 shows the implementation of the previous function ‘isEven’ from Listing 2.3:

Listing 2.4: Example of white box testing: function isEven

```
int isEven(int number) {
    if (number % 2 == 0) {
        return 1;
    } else {
        return 0;
    }
}
```

Based on the function's internals, test cases can now be designed to properly cover the test object, using e.g. control-flow oriented testing techniques as presented in Section 2.2.5.

The pros of whitebox testing are:

- The possibility to test subroutines of the system and internal functionalities, increases the chance to find bugs in the implementation, as the implementation itself is directly visible.
- Whitebox testing results in an easier automation of test case generation, as internal nodes and decision paths are known and it is therefore possible to derive test cases from the implementation.
- There is no need to have detailed specifications[68], as the system internals are directly accessible. This allows for testing of systems which are not documented/specified in detail.

The cons of whitebox testing are:

- Even if every sub path behaves correctly, it is not guaranteed that the SUT behaves according to the specification, as the whitebox test does not necessarily test against requirements.
- Depending on the grade of detail, the testing process can be costly, if, e.g., the program has many paths to execute.

Deciding whether to test the SUT using black box or white box testing is often a question of budget vs. thoroughness. As black box testing does not require in-depth analyses of the test object's internals, it can sometimes prove less expensive. On the other hand, it can be sensible to use white box testing to verify the correct functionality of complex procedures, where it would be difficult to sufficiently cover the SUT using black box testing techniques. Due to the possible grade of detail, this can be very expensive.

2.3 Penetration Testing

'Penetration testing can be defined as a legal and authorized attempt to locate and successfully exploit computer systems for the purpose of making those systems more secure. The process includes probing for vulnerabilities as well as providing proof of concept attacks to demonstrate the vulnerabilities are real' as Engebretson[27] states.

Henry[40] states that 'penetration testing is the simulation of an attack on a system, network, piece of equipment or other facility, with the objective of proving how vulnerable that system or "target" would be to a real attack'.

The above definitions conclude that a penetration tester puts him-/herself in the shoes of an attacker and tries to find vulnerabilities in the system before a real attacker does. It therefore helps to harden the system against real-world threats. Penetration testing includes the use of various tools and techniques and 'thinking outside the box' to achieve the goal of penetrating the attack target.

2.3.1 Classification

The BSI suggests a classification of penetration tests as follows [43]:

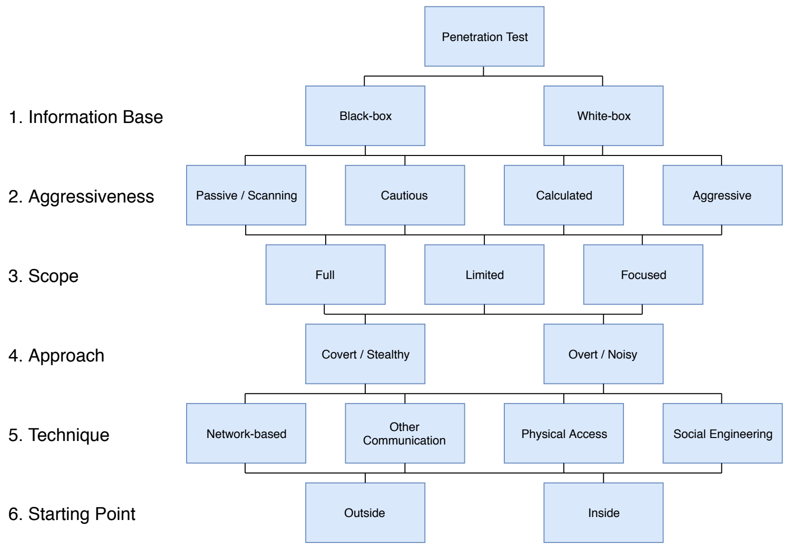

Figure 11: Classification of Pentests, rebuilt after BSI[43]

Classifying the penetration test can help to plan the test accordingly and clarify the planned approach for everyone involved and sharpen the focus on the pentest's goal. The following sections describe the different criteria.

1. Information base

The category Information Base defines the penetration test as a white box or black box test. As described in Section 2.2.8, in a white box test the tester has knowledge of the SUT's internals. A black box test sees the test object as black box with no insides whatsoever.

2. Aggressiveness

High aggressiveness can prove very efficient, but bears the risk of being detected by the system's administrators or an intrusion detection system, thus triggering countermeasures, or the risk of damaging the system and/or its data. This would be especially problematic with productive systems. An approach too low in aggressiveness however, can lead to the missing of important information and prove inefficient.

3. Scope

The scope defines how much of the system is tested. A full test scope covers the whole system. Limited or focused tests cover only parts of the system defined as relevant.

4. Approach

The kind of approach tells the penetration testers if their approach has to be stealthy and covert in order to avoid detection, or if that is not an issue.

5. Technique

Depending on the target, different techniques must be applied. A typical attack would be a network-based attack using the TCP/IP stack, but other communication channels like Bluetooth, phone lines, etc. can also be attractive targets. Furthermore, there are attacks where the attacker tries to get physical access to the target, and social engineering attacks where the attacker tries to exploit a possible weak link in the security chain: humans.

6. Starting point

The starting point defines whether the attacker starts from the inside, e.g. the local network, therefore avoiding network firewalls and similar protection mechanisms, or from the outside, e.g. over the internet.

2.3.2 Phases of a Penetration Test

From an organisational point of view, the whole process of penetration testing can be described in phase models, of which various exist.

Engebretson[27] describes the Zero Entry Hacking (ZEH) methodology, a four-step penetration testing model.

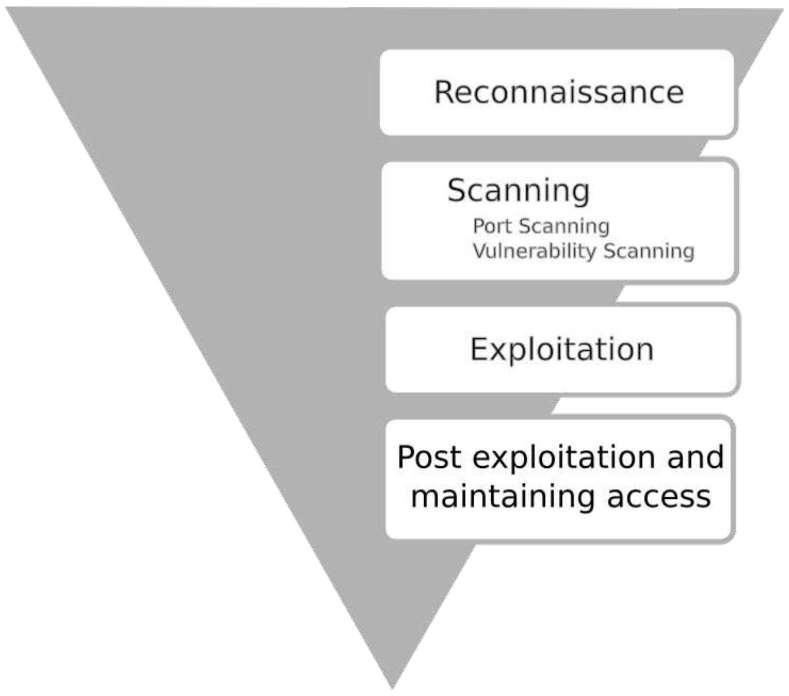

Figure 12: Zero Entry Hacking methodology, after Engebretson[27]

The four-phase model is presented with a triangle, symbolising the journey from very broad and general information gathering to very specific vulnerability exploitation techniques:

1. **Reconnaissance**: in this phase, the focus lies on information gathering. Every little piece of information about the target can prove invaluable in later phases.
2. **Scanning**: after the information gathering phase, the concrete attack targets and attack vectors are established through scanning techniques like port scanning and vulnerability scanning.
3. **Exploitation**: the exploitation phase consists of applying different exploitation techniques and tools to the previously established attack vectors.
4. **Post exploitation and maintaining access**: after successful exploitation of a target, sometimes it is necessary to ensure permanent access to the system. Further post-exploitation tasks include detailed reporting of the findings and eventually cleaning up changes made.

Scarfone et al.[79] from the NIST present a further penetration testing methodology, shown in Figure 13.

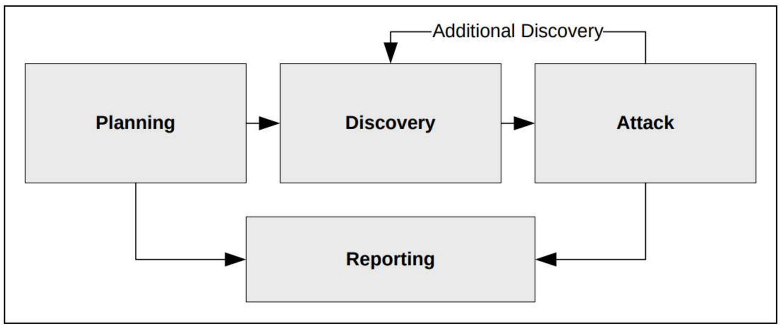

Figure 13: Penetration Testing Methodology, after NIST [79]

The penetration testing methodology consists of four steps[79]:

1. **Planning**: in the planning phase, testing goals are set, management approval is fixed and documented and further arrangements for the testing process are settled.

2. **Discovery**: the discovery phase consists of gathering information about the SUT and the identification of existing vulnerabilities by means of comparison with vulnerability databases.

3. **Attack**: the attack execution includes verifying the identified vulnerabilities. Successful exploitation verifies the security weakness. A successful attack can grant extended access to the system and enable the tester to gather further information through re-executing the discovery phase.

4. **Reporting**: in parallel to the previous three phases, reporting takes place. In the planning phase, an assessment plan is developed. In the discovery and attack phases, findings and logs are documented. The report usually includes a description of the found vulnerabilities, a rating of their risks and guidance on how to mitigate the given weaknesses.

While the ZEH methodology focuses on the actual security testing work and splits it into phases, the NISTs work has a more wholistic approach and also includes the planning of the penetration test, including the clarification of legal questions and a reporting phase to properly communicate the tester's findings.

Comparing penetration testing, using the example of the ZEH methodology[27] shown in Figure 12 and the NIST methodology[79] shown in Figure 13 and functional software testing, exemplified by Spillner[86] and shown in Figure 7, some similarities are visible.

Spillner's 'Planning and Control' phase has the same goal as the 'Planning' phase in the NIST method: to lay the organisational foundations for the test to execute. 'Analysis and Design' resemble the ZEH phase of 'Reconnaissance'/'Scanning' and the NIST phase of 'Discovery', in which the objective is to analyse the SUT and prepare for testing. The

'Implementation and Execution' phase of Spillner includes the test execution itself and is paralleled by ZEH's 'Exploitation' and NIST's 'Attack' step. Reporting of the results is then done in Spillner's 'Evaluation and Reporting' step and can be found in ZEH's 'Post exploitation and maintaining access' or NIST's 'Reporting' phase. Regardless of the similarities between functional software testing and penetration testing, there remains a big difference in the objective: in functional software testing, the goal is to test if a functionality does work as intended, whereas the security test does test if there is an aspect of the system that does not work as intended, or if there are any unintended side effects, which can be even harder to detect[93].

2.3.3 Ethics

Penetration testers might get access to very sensitive material on clients' systems, and eventually gain knowledge of vulnerabilities that can be used to harm a business or person and in general work with tools and techniques that have the power to be used for illegal and/or unethical actions. Due to this nature of penetration testing, the tester is always confronted with ethical considerations. The difference between a hacker who wants to penetrate a system illegally, a so-called 'black hat', and one who works in accordance with the attacked instance, a so-called 'white hat', is the purpose of the tester's actions. *The end justifies the means.*

'The field of ethics (or moral philosophy) involves systematizing, defending, and recommending concepts of right and wrong behaviour' as defined by Fieser[30]

Darwall practically defines ethics as 'enquiry into what we ought to desire, feel be, or do' [20].

Schopenhauer argues, however that insight into ethical principles does not automatically lead to compliance with discussed principles. Additional incentives or enforcements are necessary. This is called the 'enforcement problem' [81].

Pierce[72] mentions several ethical questions a penetration tester's work might be subject to, of which an excerpt shall be presented here:

- *As a system's security is not provable (only its insecurity) [34], a tester should make it clear to the SUT's owner that a penetration test with no findings does not necessarily mean that the system is secure. That is a common false expectation. Penetration testers do not prove security, they prove only insecurity.*
- *Testing should only be performed with the written permission of the client. This is a necessary step to separate the black hat hacker from the security expert.*
- *A penetration tester should deeply understand the tools and techniques he/she is using to fully grasp the extent of every performed action. Solely relying on automated 'black box' tools would be a mistake in this sense.*

- *The results of a penetration test must be subject to non-disclosure, Otherwise a client and its reputation might be at risk of damage. However, it is the opinion of the writer of this thesis that it depends on the setting and agreements with the client. In publicly used libraries, for example, full disclosure can be beneficial to the users to understand the risk.*
- *The tester should immediately notify the client about severe vulnerabilities, especially when the results of exploitation may lead to serious damage, like danger to human life.*
- *The outcome of social engineering actions should only be published in summarised form to not implicate individual employees and possibly cause harm to their reputation or employment.*

All ethical questions involved in penetration testing circle around the tester's personal integrity.

2.4 E-Health

Section 2.4.1 will give an introduction to eHealth and further elaborate the term using various examples.

2.4.1 Introduction

The definition of the term eHealth seems to vary in scientific literature – a literature study by Oh[61] found 51 unique definitions. A broad definition is given by Eysenbach [28]:

> 'E-health is an emerging field in the intersection of medical informatics, public health and business, referring to health services and information delivered or enhanced through the Internet and related technologies. In a broader sense, the term characterizes not only a technical development, but also a state-of-mind, a way of thinking, an attitude, and a commitment for networked, global thinking, to improve health care locally, regionally, and worldwide by using information and communication technology.'

Wilson et al.[96] discuss 'eHealth, like eGovernment and eCommerce, is about placing citizens at the centre of the circle and easing their interaction with the wide range of people who look after their health needs'. They further explains that in the 1960s, health informatics and bio-medical computing found a place only with academic interest groups. An exemplary use was the calibration and dosage calculation in radiotherapy to optimise the dosage of radiation to the right parts of the body. Since the 1970s, computer technologies were also used to keep track of patient notes, bed occupation and other planning tasks. As many stand-alone applications were developed to solve the challenges, the need for integrated solutions that follow a patient's interactions with the health service providers arose.

However, eHealth solutions are not limited to medical practitioners. With the rise of eHealth systems, people in the EU-28 also seem to be literate in digital health. According to a Eurobarometer survey from 2014[31], 59% of people used the internet to search for health-related information in the last 12 months.

2.4.2 Examples

For better understanding of the nature and implications of eHealth, the following example shall give an overview over the eHealth strategy in Austria:

Pfeiffer[73] recommends an eHealth strategy for Austria and formulates its vision as an integrated management of citizen's health, using information and communication technologies to support the processes of every actor in the field of healthcare, under special consideration of data protection and data security.

To implement this vision, he further mentions the following elements in the recommended eHealth environment:

- Elektronische Gesundheitsakte (ELGA) is a project to standardise, digitalise and centralise patients' health data [26].
- The e-card, a card to uniquely identify a patient.
- A directory of patients.
- A directory of healthcare service providers.
- An eHealth portal as central point of access to information and eHealth applications, e.g. eMedikation, a service that centrally stores a patient's medication list.
- Comprehensive support for information and communication technology processes.
- Access to telemedical services.
- Decision-supporting systems.
- Tools for analysis of data for research and science, planning, controlling and observation of activities in the health care system.
- Technological and organisational measures for data protection and data security.

According to Pfeiffer[70], a significant future challenge is to guarantee the semantic interoperability of the systems in the eHealth environments through international standards, like ICD-10 or SNOMED-CT. Such a standard would support international healthcare collaborations.

To further grasp the meaning and various types of eHealth, a small excerpt of further examples shall be given:

- COMMUNITY HEALTH ENGAGEMENT SURVEY SOLUTION (CHESS) is a mobile app that collects data from four health risk factors: alcohol, food, physical activity and tobacco, entered by the participants through surveys. The data is then used as basis for the planning of health promotion actions and disease prevention programmes [89].
- GET CONNECTED is a website to raise awareness about Sexually Transmitted Infection (STI)s and guides people on their way to the next testing facility [89].
- The ACT@Scale project, funded by the European Commission, evaluates best practices in the field of care coordination and telehealth and how they can be scaled up from pilots and experiments to routine management processes [2].

As the examples in this chapter indicate, eHealth is not only a collection of tools, but also includes the establishment of infrastructure, legal frameworks, standards, education and awareness.

The processing of personal data as sensitive as health data also has certain requirements to meet in the field of data protection and data security (see Section 2.1.3). Given the sensitivity of the data and the need to prevent a certain risk of data theft or data misuse, the question for further research arises, whether an individual should be forced to participate in a state's highly integrated eHealth environments or not. There might be individuals who are not comfortable with the centralised storage of their health data, rendering it accessible by medical practitioners. Strict access rules, maybe even controlled by the patient, could help to build trust.

2.5 Large IT-Infrastructures

Sections 2.5.1 and 2.5.2 will introduce the meaning, characteristics and issues of large IT infrastructures.

2.5.1 Introduction

With increased sizes of IT infrastructures, new needs, requirements and challenges arise, e.g. the need for extensive access control mechanisms. Control of individual instances, be they computers or people, becomes more and more difficult and small issues that could easily be neglected in small networks scale up to serious issues in large IT infrastructures. A significantly higher dimension of such infrastructures also brings the challenge of different requirements for organisational processes and organisational security measures, e.g. an Information Security Management System (ISMS). The ISO[45] defines an ISMS as follows:

> ‘an ISMS is a systematic approach for establishing, implementing, operating, monitoring, reviewing, maintaining and improving an organization’s information security to achieve business objectives. It is based on a risk assessment and the organization’s risk acceptance levels designed to effectively treat and manage risks.’

In other words, it is at the core of organisational processes to manage information security.

Section 2.5.2 summarises some of the characteristics and issues that might occur in large IT infrastructures.

2.5.2 Characteristics & Issues

Due to the bigger size and often heightened complexity, characteristics in large IT infrastructures can differ from smaller networks. Some of the characteristics might be:

Size: The most obvious characteristic of large IT infrastructures is their size. Whereas in small-sized networks, there might be handful of members, in larger corporate-sized networks, there can be thousands, or even millions of participants.

Human Factor: Due to the larger size, people working together differ in knowledge and behaviour and often work in more specialised areas.

Complexity: The higher number of participants can result in higher complexity. System complexity stems from the number and type of relationships between the system’s components and between the system and its environment, as Sommerville et al.[84] state. With larger businesses, larger infrastructures and more people comes the need to distribute responsibilities, as one person can only do a limited amount of work. Distributed responsibilities between employees increases communication effort, as information and decisions need to be collected from different people. Larger companies, especially multinational ones, have an organisation and possibly projects, which span across several countries, with all the complexity that comes with it. Increased communication effort arises through difficulties including different time zones, cultures, work ethics and languages. Infrastructure requirements, e.g. network infrastructures, face challenges like different geographic locations, or bringing together infrastructure security measures of different locations. For large companies with large IT infrastructures, various services and tools need to be able to handle the larger magnitude of participants. This raises the question, whether it is cheaper to outsource a service or tool to a supplier which is specialized in the given field and might offer a cheaper solution than to host the service or tool in-house. Outsourcing, however, brings other questions that need to be considered, e.g.: what level of data security can be guaranteed? What happens, when the service is unavailable?

Given the characteristics of large IT infrastructures, various challenges arise. A sample shall be given in the following list:

Size: In sizes of higher magnitude, simple problems can occur in larger quantities and thus prove more expensive to get rid off, e.g. if the newest operating system update contains a malfunction, which causes the malfunction to appear on thousand computers simultaneously. The administration of such infrastructures generates a certain amount of overhead including administrative systems that need to be administered themselves.

Human Factor: It cannot be expected that everyone in the network has the same level of knowledge, so an amount of support needs is given on the human side. The human factor can bring further challenges. If, for example, an employee uses easily guessable passwords, this can pose a severe security risk. An attacker could guess the password and gain access to the user's account.

Security: The need for additional security measures arises in such environments. Not everyone should be able to do everything, so an authentication and authorisation system is needed. As huge infrastructures with many participants imply a big attack surface, appropriate measures need to be taken against attacks from the outside and inside. E.g., a computer virus infection of the whole system could prove disastrous, if it renders the infected computers useless.

Complexity: Johnson[47] defines complexity as 'phenomena which emerge from the collection of interacting objects'. Enhanced complexity requires simplification in order to be able to manage the system. To maintain control and solve many problems before they appear, usually much stricter policies are installed and applied on the organisational level, e.g. certifications.

Testability: As Schanes et al.[80] state, new systems often cannot be tested in the production system for fear of disturbances of processes in the live systems. A perfect replication of the live system can prove too expensive in large IT infrastructures. Sometimes, however, it can be insufficient to test new systems in dedicated testing environments, which constitutes a problem.

Time & Budget: In large IT infrastructures, which are often present in the corporate field, there is usually a given budget and time frame in which projects need to happen or issues need to be resolved. This results in a certain cost and time pressure to the actors involved and can lead to the need to compromise between cost, time and quality. Neglecting the management of those constraints could lead to the loss of cost control.

The list of characteristics and issue depict that large IT systems imply different measures and ways of thinking. In particular, requirements for security need consideration, as failure to do so in large and complex IT infrastructures can result in costly issues, like the breakdown of components many hundreds of people need to work. Extensive measures are needed to mitigate security issues, however, when it comes to high cost, a valid consideration can be to deliberately accept a security risk, if the anticipated damage is lower than the cost of fixing the issue. Considering all the characteristics and issues that come with large IT infrastructures, administrative and security measures must not interfere with people's efficiency too much, as this would lead to a competitive disadvantage.

2.5.3 Example: German Health Telematik Infrastructure

Telematik is a field which combines telecommunications and computer science and was first discussed by Nora and Minc[60]. The German Health Telematik Infrastructure (TI) is an infrastructure which connects all participants in the health system: physicians, dentists, psychotherapists, hospitals, pharmacies, public health insurance companies and patients, with the goal to centralise, simplify and accelerate the communication between these stakeholders[90]. The law that regulates the introduction and use of the TI is the German 'Gesetz für sichere digitale Kommunikation und Anwendungen im Gesundheitswesen sowie zur Änderung weiterer Gesetze'[37], which has been issued on December 21, 2015.

As of July 1, 2019, there are approx. 73 million health insured people in Germany who might use the TI[88].

The TI consists of several components and applications, as described by the Kassenärztliche Bundesvereinigung (KBV)[90] and the gematik[36] in Sections 2.5.3 and 2.5.3.

Components

The *Konnektor* is a special router which has been certified by the BSI and establishes a secure connection to the TI to enable the user to access the TI applications.

The *'elektronische Gesundheitskarte (eGK)'* is a German insurance card, including a photo and further data which identifies the insured person[85].

The *'elektronischer Heilberufsausweis (eHBA)'* is a card which identifies the doctor, dentist, psychotherapist, pharmacist and members of other health professions. It is not mandatory to connect to the TI, but is necessary for some of the applications and for the creation of electronic signatures.

With the *Security Module Card Type B (SMC-B)*, a health profession member can authenticate as such and use the Konnektor to connect to the TI.

In order to read eGK, eHBA and SMC-B a card terminal, connected to the Konnektor is necessary.

With these components it is possible to connect to the TI and use the services provided there. This gives an example of the degree of complexity that is necessary to securely connect millions of users to a central eHealth (see Section 2.4) infrastructure.

Applications

The TI includes several applications, which it offers, or will offer, to their users. It is possible that in the future, additional applications will be created.

The Versichertenstammdatenmanagement (VSDM) is an application for the management of a patient's personal data, e.g. home address. Since July 1st, 2019, it is mandatory for every practitioner in the health profession to use the VSDM[37]. When a patient sticks the eGK in the card terminal at the doctor's office, it automatically checks the insurance data for changes and eventually updates the data on the eGK. This way, the data on the card stays up-to-date.

Using the Notfalldatenmanagement (NFDM), a patient can let a doctor store emergency-relevant data on the eGK. This includes data like chronic diseases, regular medication, allergies, further medical information, and additional contact data of persons to contact or doctors[35]. This data shall aid in the case of emergency.

The elektronische Medikationsplan (eMP) is an application which allows the patient to let the doctor store medication data on the eGK, if the patient takes three or more medications permanently. This way, possible issues related to medications interacting with each other, shall be mitigated.

The elektronische Patientenakte (ePA) gives the patient the possibility to store clinical reports, data about medical outcomes and therapy measures. This enables the doctor to see the patient's history and adapt accordingly. All this, however, only with the patient's explicit consent.

The above descriptions of the TI show the magnitude such large infrastructures can grow to and give a glimpse of how such large eHealth infrastructures are constructed.

CHAPTER 3

Optimisation of Security Test Concept Efficiency

Current issues of penetration testing are identified in Section 3.1. Section 3.2 will introduce the difference between efficiency and effectiveness. Section 3.3 will analyse various optimisation approaches and elicit an optimisation method described in Subsection 3.4, which will conclude the section about establishing the optimisation method.

3.1 Issues of Current Penetration Tests

There are several issues, which need consideration when dealing with penetration tests. They shall be separated in legal issues, planning issues, execution issues and issues after penetration test execution.

3.1.1 Legal Issues

According to BSI[11], there are three aspects to consider with legal issues in the field of penetration testing:

- Legal reasons can motivate a company or public authority to perform execution tests. If, for example, a regulation states, that a system as part of critical infrastructure must conform to a certain norm and this norm states that the system must undergo regular penetration tests, then the system is subject to penetration testing. The EUGDPR[67] makes demands to state-of-the-art security of processed data, which might also encourage a data processing business to facilitate penetration testing.
- When conducting penetration testing activities, there are certain legal regulations and principles, which need to be considered and clarified beforehand. One example

would be to define that the penetration tester is (not) liable, if production data is lost.

- Especially in the planning phase, it is essential to lay the legal foundations in form of a contract between the client and the penetration tester.

As these legal aspects are the basis of the testing work, they need to be clarified with the client and observed during the penetration testing process, in order to prevent unwanted ramifications.

3.1.2 Planning Issues

During the planning phase of the penetration test, aside from the legal issues, some other organisational and technical questions need to be answered.

In addition to the classification criteria mentioned in Section 2.3, the BSI[11] adds further requirements and questions to consider:

Organizational Questions:

- Who, apart from the client, will be affected either directly or indirectly by the penetration test?
- Have the liability risks received appropriate consideration?
- What needs to be considered in respect of the time of testing?
- What needs be done in the event of system failure or other emergency?
- Which of the client's employees are affected by the penetration test?
- How much time and cost will the penetration test involve for the client?
- How much time and effort will the penetration test require of the tester?

Personnel Requirements:

- Knowledge of system administration/operating systems
- Knowledge of TCP/IP and, if applicable, other network protocols
- Knowledge of programming languages
- Knowledge of IT security products such as firewalls, intrusion detection systems
- Knowledge of how to handle hacker tools and vulnerability scanners
- Knowledge of applications/application systems

- Creativity

Technical Requirements:

- Access to public networks: Access to the internet or the public telephone network is an important precondition for performing the penetration test since most attacks are launched over these communication channels. A sufficiently high-capacity internet connection should therefore be available. Here it is important to note that vulnerability scanners in particular require a high bandwidth. The efficiency of testing therefore depends for one thing on the available line capacity.
- Availability of suitable auditing tools: The penetration tester must have suitable tools at his disposal for performing the tests. Many of these tools can be downloaded from the internet free of charge. Tools such as vulnerability scanners, however, often attract extremely high royalties (usually depending on the number of IP addresses to be scanned). An efficient test requires the “right” tools rather than large numbers of tools. The tester knows the effects and side-effects of the tools and is often able to assess a large number of results quickly and differentiate false statements from true ones.
- Local test network: The various tools must be tested in a local test network before use in a real penetration test. These kinds of tests also allow the penetration tester to familiarize himself with hacker tools and vulnerability scanners and with the results they produce. If the systems of the test network are suitably configured, they also allow vulnerabilities in the systems to be tested and verified.

Neglecting planning requirements or questions can lead to unpleasant surprises during the execution phase.

3.1.3 Execution Issues

In functional testing, the tester has a requirement to test. If the test object’s behaviour fulfils the given requirement, the test was successful and the requirement proven to be fulfilled. If it does not, the opposite is the case (see also Section 2.2.1). If, for example, a requirement states, that a text should appear when a button is clicked, and upon clicking the button the text actually appears, the requirement is fulfilled. If the text does not appear, or another text appears, the requirement is not fulfilled and the functionality is therefore incorrect. There is a defined required behaviour and a deviation from it is deemed a violation of that requirement.

In security testing, it cannot be proven that a system is really secure, as stated by Geer[34]. For a SUT to be seen as secure would require the total absence of security issues, which would require the knowledge and enumeration of all possible vulnerabilities. This, however, is not possible as there might be several yet undetected, unknown or unanticipated security issues. When testing a system for vulnerabilities, the tester actually tests for the system's insecurity. If a security issue is found, the SUT is proven to be insecure. This leaves the security tester with the following possibilities: test for known vulnerabilities and try to find unknown ones and is in contrast to functional testing, where the functionality can be tested directly.

This bears the risk that without a plan, a security tester could endlessly test on a part of the SUT, going ever deeper into the matter, without testing the other parts of the SUT and use up the testing budget without covering the test object sufficiently.

To prevent this, a test concept, or security test concept in this case, guides the tester through the test execution. To maximise the test coverage in the given budget, the following optimisation methods have been identified and will be described in detail in Section 3.3:

1. Test suite reduction to reduce redundant test cases in the security test concept.
2. Test case automation to reduce execution cost of recurring test executions.
3. Test case prioritisation to define an order of test execution and make sure test cases with higher priority are executed first. This also helps to make sure high priority test cases are executed within a given testing budget.

3.1.4 After-Execution Issues

To prevent issues in the phase after penetration test execution, the NIST[79] suggest:

> 'Following the execution phase — whose findings are expressed in terms of vulnerabilities — the organization should take steps to address the vulnerabilities that have been identified ... First, final analysis of the findings should be performed, and mitigation actions developed. Second, a report should be developed to present the recommendations. Lastly, the mitigation activities should be carried out. Many of the actions presented in this section may occur outside of the testing process itself—for example, as part of a risk assessment that utilizes testing results.'

The Penetration Testing Execution Standard (PTES)[92] suggests a report structure consisting of two sections including the following points, in order to tackle the issue of a twofold target audience - management to get a situation overview and engineers to fix found issues:

1. **Executive Summary**
    - Background: Explains what the purpose, the agreed terms, and the objective of the test were.

- Overall Posture: The overall effectiveness of the test and brief descriptions of failures discovered.
- Risk Ranking: An overall risk scoring, using a method agreed upon in the planning phase.
- General Findings: The general findings provide a synopsis of found issues in a basic and statistical format.
- Recommendation
  Summary: This section provides a high level understanding of the tasks need to mitigate the found issues.
- Strategic Roadmap: The strategic roadmap is a detailed plan for remediation of found security issues and creates a path of actions to follow.

2. **Technical Report**

- Introduction: An introduction gives basic knowledge about the general circumstances and parameters, the test was executed in.
- Information Gathering: Presents the information that could be gathered about the SUT.
- Vulnerability Assessment: This describes the found vulnerabilities.
- Exploitation: The exploitation acts as a vulnerability confirmation and explains details about the steps taken.
- Post Exploitation: The post exploitation section outlines the consequences of successful exploitations for the business.
- Risk/Exposure: This section presents a quantification of detected risks.
- Conclusion: A round-up of above points.

This concludes the questions, considerations and issues in penetration testing in the areas of legal issues, planning issues, execution issues and issues after penetration test execution. The points mentioned in this section can not claim to be complete by any means, as there are always unforeseen possibilities and events that need adaption to the given situation.

3.2 Efficiency vs. Effectiveness

For optimisation of efficiency of a security test concept, it is necessary to know exactly what efficiency is and what distinguishes it from effectiveness.

Efficiency and effectiveness are two terms, which are often used unclearly, indistinctly or without being sufficiently defined. They reflect the definitions from the field of software quality assurance in section 2.2.2 of *verification*, meaning the check if a product is built the right way, and *validation*, meaning the check if the right product, according to the customer's needs and wishes, is being developed.

Drucker[23] distinguishes between efficiency and effectiveness the following way and discusses the confusion between them as a problem in business:

> 'It is fundamentally the confusion between effectiveness and efficiency that stands between doing the right things and doing things right. There is surely nothing quite so useless as doing with great efficiency what should not be done at all. Yet our tools—especially our accounting concepts and data—all focus on efficiency. What we need is (1) a way to identify the areas of effectiveness (of possible significant results), and (2) a method for concentrating on them.'

More formal definitions are given by the norm ISO 9000:2015 [63]:

Efficiency: *Relationship between the result achieved and resources used.*

This can be defined by the following mathematical term:

$$\frac{Result}{Resources}$$

Effectiveness: *Extent to which planned activities are realized and planned results are achieved.*

Analogously, the following fracture defines effectiveness:

$$\frac{Result}{Objective}$$

Imagine a woodcutter who is ordered to cut down specific trees in a forest. He might prepare for the task by sharpening his saw and axe and determine the most suitable tree-felling technique. The trees are cut down very efficiently, rendering the most outcome using the fewest resources. However, after the procedure the woodcutter finds out, that the wrong trees were cut down, maybe even in the wrong forest.

In software development especially, the topic is of importance and can prevent or cause significant project cost. The project team might deliver code in great quality and do so very efficiently and quickly. However, it proves very expensive to find out that the delivered software does not fit the requirements and the customer's needs.

Sometimes less expensive, but still problematic is the case where the woodcutter is working on the right trees. When asked why he/she is not sharpening the saw, the response is: 'there is no time for this; I need to work on cutting down the trees', as the woodcutter continues to work with a blunt saw. In the long run, this causes more cost and/or time delay than necessary.

The first example is the neglect of effectiveness for efficiency's sake, resulting in the wrong results. The second example is the neglect of efficiency for effectiveness, resulting in work on the right path, but without measures of improving efficiency. Both cause increased cost. For successful projects, both effectiveness and efficiency are equally necessary. Effectiveness comes first to ensure that efficiency is targeted in the right direction.

In security testing, effectiveness means planning and executing the right tests on the right SUT. Efficiency in this context would mean using fewer resources, e.g. time, human resources or computational resources, for executing tests. Efficiency without effectiveness would lead to ever more quickly executed test suites, optimised by sophisticated productivity methods or techniques, missing, however, any guarantee that what is highly efficiently tested, is the required system under test. The other way around, effectiveness without the necessary amount of efficiency can lead to the burning up of the allocated security testing budget very quickly. This could be because of slowness of the tester himself/herself, or because of getting too deep into one part of the system or one part of the test and loosing oneself.

The latter phenomenon is due to the fact that in security testing, one could go ever deeper and test new approaches in ever greater detail, without ever proving the system's security. Geer and Harthorne express this peculiarity the following way [34]:

> 'Penetration testing is the art of finding an open door. It is not a science as science depends on falsifiable hypotheses. The most penetration testing can hope for is to be the science of insecurity - not the science of security - inasmuch as penetration testing can at most prove insecurity by falsifying the hypothesis that any system, network, or application is secure. To be a science of security would require falsifiable hypotheses that any given system, network, or application was insecure, something that could only be done if the number of potential insecurities were known and enumerated such that the penetration tester could thereby falsify (test) a known-to-be-complete list of vulnerabilities claimed to not be present. Because the list of potential insecurities is unknowable and hence unenumerable, no penetration tester can prove security, just as no doctor can prove that you are without occult disease.'

This is why, without clear constraints, the tester could run into danger of going into greater detail than is necessary and using up too many resources, thus being inefficient.

3.3 Theoretical Optimisation Approaches

As explained in Section 3.1, security tests can grow quite extensively, if not constrained properly. This makes the topic of penetration testing very interesting for optimisation approaches. With the goal to establish an optimisation method, this chapter will guide through the process of doing so. To develop a suitable optimisation method, three test suite optimisation techniques shall be analysed: test suite reduction in Section 3.3.1, test automation in Section 3.3.2 and test case prioritisation in Section 3.3.3. The three optimisation approach sections give an introduction into the topic, present examples of the technique and conclude with a suggestion on how to improve the test suite's efficiency in execution.

3.3.1 Test Suite Reduction

Test Suite Reduction techniques can significantly reduce a test suite's size and, therefore, help save test execution time, test data management efforts and consequently costs, as stated by Khan[74].

Harrold et al.[39] define the problem of selecting a SUT's representative set of test cases as follows:

Given: A test suite TS, a set of test case requirements $r_1, r_2, ..., r_n$ that must be satisfied to provide the desired testing coverage of the program, and subsets of TS, $T_1, T_2, ..., T_n$, one associated with each of the r_i's such that any one of the test cases t_j belonging to T_l can be used to test r_i.

Problem: Find a representative set of test cases from TS that satisfies all of the r_i's.

Harrold further continues, that a representative set of test cases that satisfies the r_i's must contain at least one test case from each T_i. Such a set is called a hitting set of the group of sets $T_1, T_2, ..., T_n$. A maximum reduction is achieved by finding the smallest representative set of test cases. However, this subset of the test suite is the minimum cardinality hitting set of T_i's and the problem of finding it is NP-complete.

So the objective of the Test Suite Reduction (TSR) problem is to find a representative reduced test suite containing a minimal number of test cases from T that satisfies each r_i at least once[74].

Khan et al.[74] selected 133 from an initial pool of 4230 studies and classified them as shown in Figure 14 and descriptions below:

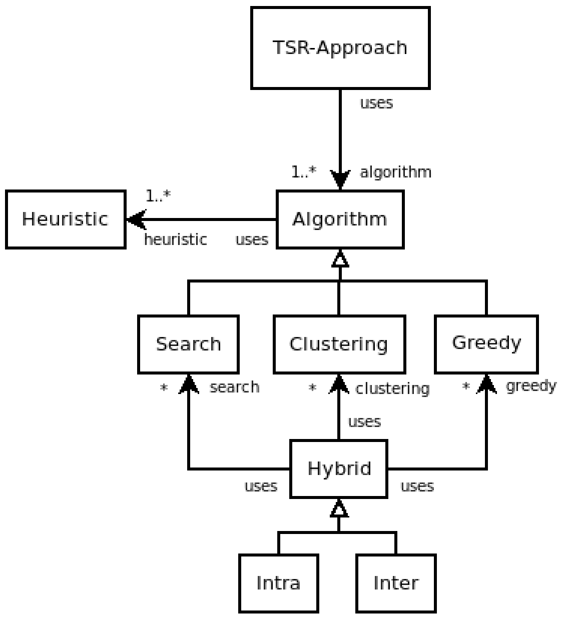

Figure 14: Conceptual diagram of TSR approaches[74]

Based on the algorithms employed, the TSR approaches were classified into four main categories: Greedy, Clustering, Search and Hybrid, which is a combination of the first three classes.

He further gives the following definitions as basis for the classification[74]:

$TS = tc_1, tc_2, ..., tc_{nts}$ is the original set of test cases, on which the TSR algorithm shall be applied and 'nts' is the number of test cases in this set.

$RS = rtc_1, rtc_2, ..., rtc_{nrs}$ is the reduced set of test cases, which is a subset of the original set. 'nrs' is the number of test cases in the reduced set, where $nrs < nts$

$Heuristic = h_1, h_2, ..., h_{nh}$ is a set of heuristics used in the TSR approach, where 'nh' represents the number of heuristics used.

$Cost = cost_1, cost_2, ..., cost_{ncost}$ is a set of cost measures and 'ncost' is the number of cost measures in the set. Cost measures could be execution time of the reduced set or cost of the algorithm's application.

$AlgorithmClass = class_1, class_2, ..., class_{nac}$ is a set of algorithms, based on which the classification is done, and which are employed for TSR. 'nac' is the number of algorithms in
$AlgorithmClass$.

$CostF(TestSuite, cost_i)$ is a function returning the test suite $TestSuite$'s cost, using a cost measure $cost_i$ from the set of cost measures $Cost$.

$EffectF(TestSuite, effect_i)$ is a function returning the test suite $TestSuite$'s effectiveness, based on an effectiveness measures $effect_i$ from the set of effectiveness measures $Effect$.

The goal of TSR can be formally defined, using above definitions, as shown in equations 3.1 and 3.2 [74]:

$$\sum_{i=1}^{ncost} CostF(RS, cost_i) \leq \sum_{i=1}^{ncost} CostF(TS, cost_i) \tag{3.1}$$

where $ncost \geq 1, RS \neq \emptyset, TS \neq \emptyset$

$$\sum_{i=1}^{neffect} EffectF(RS, effect_i) \leq \sum_{i=1}^{neffect} EffectF(TS, effect_i) \tag{3.2}$$

where $neffect \geq 1, RS \neq \emptyset, TS \neq \emptyset$

This means, that the sum of the cost of the resulting test suite is less, or equal to the cost of the original test suite. To reduce a test suite's cost is of course the goal of test suite reduction. However, as a consequence of reducing the test suite, the resulting test suite's effectiveness will also be less or equal to the original test suites's one. If test cases are removed from a test suite, the test suite cannot be more effective than before, as it can now cover only less or equally much of the SUT (also see Equation3.5).

In the following paragraphs, the four classifications of TSR approaches are defined in detail.

Greedy-Based Approach

A greedy algorithm tries to find the local optimal solution by determining which partial solution is best in a given step, e.g. offers the highest SUT coverage. The following definitions will define greedy-based TSR approaches formally[74]:

$Coverage = coverage_1, coverage_2, ..., coverage_{ncoverage}$ is a set of coverage criteria and $ncoverage$ is the number of coverage criteria. $Coverage \subset Effect$. $Coverage$ is a subset of $Effect$ (the set of effectiveness measures).

$Coverage(TS, Criterion)$ is the function that provides TS's coverage in percent, based on the criterion $Criterion$ from $Coverage$, the set of coverage criteria.

A binary relation matrix $Satisfy(TS, Criterion)$ can be used to describe if a test case satisfies one or more elements of a set of criteria. In tabular form, rows would represent the test cases and columns the coverage criteria.

$$Satisfy(TS,\ Criterion) = (tc,\ c) \mid tc\ satisfy\ c,\ tc \in TS\ \wedge\ c \in Criterion$$
$$\begin{cases} 1, & \text{if } tc\ satisfy\ c \\ 0, & otherwise \end{cases} \tag{3.3}$$

A greedy-based test suite reduction approach uses a greedy algorithm from $class_i$ from $AlgorithmClass$, that uses one or more greedy heuristics from $Heuristic$ to obtain RS such that conditions from equations 3.4 and 3.5 hold:

$$nrs < nts \tag{3.4}$$

$$\sum_{i=1}^{ncoverage} Coverage(RS, coverage_i) \leq \sum_{i=1}^{ncoverage} Coverage(TS, coverage_i), \quad where\ ncoverage \geq 1 \tag{3.5}$$

So the steps a greedy algorithm takes are as follows:

1. The function $Selection(tc_i)$ delivers the best candidate test case based on the maximum of covered entries of the binary matrix $Satisfy(TS, Criterion)$. A random test selection strategy is used when it comes to a tie between two test cases:
$$\forall tc_j \in TS \wedge i \in \mathbb{N}^+, i \leq nts : Selection(tc_i) = Satisfy(TS, Criterion) \tag{3.6}$$
2. The entries covered by the test case (tc_i) in $Satisfy(TS, Criterion)$ are marked as 0 (satisfied), tc_i added to the result set RS and removed from the TS:
$$RS = (RS \cup tc_i)\ and\ TS = (TS \backslash tc_i) \tag{3.7}$$
3. If all the entries in $Satisfy(TS, Criterion)$ are marked as 0, $TS = \emptyset$, and nrs < nts, return the RS, otherwise continue with step 1.

Clustering-Based Approach

The clustering-based approach uses cluster analysis to reduce a given test set. Given a set $TS = S_1, S_2, ..., S_k$, where each S_i is a subset of TS resulting from cluster analysis. The test cases in each S_i are similar to each other, with similarity being expressed by a similarity measure like Jaccard index, as described by Jaccard[46] or Levenshtein distance, as described by Levenshtein[53]. Each S_i is different from each other subset, based on a dissimilarity measure like Euclidean distance, as stated by Dickinson[22]. Finally, a test case is sampled from every S_i and returned as the resulting set RS[22].

The following definitions will lay the base for defining the steps for the clustering-based analysis[74]:

$SubsetTS = subset_1, subset_2, ..., subset_{nsubset}$ is a set of subsets of TS and $nsubset$ is the number of subsets.

$SimilarityMeasure = sm_1, sm_2, ..., sm_{nsm}$ is a set of similarity measures and nsm is the number of similarity measures. The similarity measures is based on cost and/or effectiveness measures.

$Similarity(SimilarityMeasure, tc_1, tc_2)$ is a function that takes a $SimilarityMeasure$ and test cases tc_1 and tc_2 and returns the according similarity between the two test cases.

$DissimilarityMeasure = dsm_1, dsm_2, ..., dsm_{ndsm}$ is a set of dissimilarity measures and $ndsm$ is the number of dissimilarity measures. The dissimilarity measure is calculated based on cost and/or effectiveness measures.

$Dissimilarity(DissimilarityMeasure, tc_1, tc_2)$ is a function that takes a $DissimilarityMeasure$, creates subsets $subset_1$ and $subset_2$ and returns the according dissimilarity between the subsets.

Using above definition, Dickinson[22] defines the following steps a clustering-based approach goes through:

1. An algorithm from $class_i$ from $AlgorithmClass$ is applied, where $class_i$ is the class of clustering-based algorithms. For every $subset_k$ in $SubsetTS$ equations 3.8 and 3.9 needs to hold.

$$\forall i, j \wedge i \neq j : Similarity(sm_1, tc_i, tc_j) \geq SimThreshold \tag{3.8}$$

   sm_1 being the first similarity measure, i and j ranging from 1 to k and $SimThreshold$ being below tc_i's and tc_j's similarity.

$$\forall i, j \wedge i \neq j : Dissimilarity(dsm_1, subset_i, subset_j) \geq DisThreshold \tag{3.9}$$

   dsm_1 being the first dissimilarity measure, i and j ranging from 1 to $nsubset$ and $DisThreshold$ being below $subset_i$'s and $subset_j$'s dissimilarity.

2. Get RS by sampling x test cases from each $subset_k$ by applying sampling algorithms and marking $subset_k$ as satisfied, such that $nrs < nts$.

3. Return RS, if $SubsetTS$ is marked as satisfied, else go to step 2.

Search-Based Approach

Khan[74] describes search-based TSR approaches as follows: for finding a reduced test set, there is a set of potential solutions in TS: $S = S_1, S_2, ..., S_k$, where k is the total number of potential solutions and can be measured as $2^K - 1$, while S_i's size ranges from 1 to n. $Fitness(solution)$ denotes a function returning a solution's fitness. Fitness is calculated based on cost and/or effectiveness measures, depending on the objective. As in clustering-based approaches, search-based approaches facilitate $SimilarityMeasures$

to determine a similarity score between pairs of test cases, leading to a diverse set of test cases. A search-based approach tries to find the best RS from the set of potential solutions S by using a search algorithm, while equations 3.10 to 3.13 need to hold:

$$\forall i : Fitness(RS) > Fitness(s_i),\ i\ ranges\ from\ 1\ to\ k \tag{3.10}$$

$$CostF(RS, cost_j) \leq CostF(TS, cost_j) \tag{3.11}$$

$$EffectF(RS, effect_j) \leq EffectF(TS, effect_j) \tag{3.12}$$

$$nrs < nts \tag{3.13}$$

Again, the cost and effectiveness of the resulting test suite are less or equal to the cost and effectiveness of the original test suite. This seems intuitive for cost, as the removal of test cases decreases cost, or in some cases cost stays the same. For effectiveness, this seems counter-intuitive, as TSR decreased effectiveness is usually not an objective in test suite optimisation. However, it is a side effect in TSR, as the reduction of test cases cannot increase effectiveness. It might increase efficiency (see Section 3.2 for the difference between effectiveness and efficiency), but effectiveness stays either the same, or decreases.

According to Harman[38], search-based TSR approaches consist of the following steps:

1. RS is initialized as empty; $RS = \emptyset$
2. For all s_i apply the fitness function, which is based on equations 3.14 and 3.15, which calculate the cost and effectiveness of all solution candidates, for every cost measure and every effectiveness measure.

$$\begin{aligned} CostF(RS, cost_i) = \sum_{i=1}^{k}\sum_{j=1}^{n} CostF(s_i, cost_j) \leq \\ \sum_{i=1}^{ncost} CostF(TS, cost_i) \end{aligned} \tag{3.14}$$

$$\begin{aligned} EffectF(RS, effect_i) = \sum_{i=1}^{k}\sum_{j=1}^{n} EffectF(s_i, effect_j) \leq \\ \sum_{i=1}^{neffect} EffectF(TS, effect_i) \end{aligned} \tag{3.15}$$

3. Cost and/or effectiveness is then compared using a cost measure and/or effectiveness measure:

$$CostF(RS, cost_i) \leq CostF(TS, cost_i) \tag{3.16}$$

$$EffectF(RS, effect_i) \leq EffectF(TS, effect_i) \tag{3.17}$$

4. Deduce the result set RS by checking equations 3.18 and 3.19:

$$\sum_{i=1}^{k}(Fitness(RS) > Fitness(s_i)) \tag{3.18}$$

$$nrs < nts \tag{3.19}$$

Hybrid Approach

Hybrid approaches combine one or more types of TSR algorithms. A distinction can be made between:

- *Intra Hybrid Techniques* employ more than one algorithms from the same *AlgorithmClass* for test suite reduction.
- *Inter Hybrid Techniques* facilitate algorithms from more than one different *AlgorithmClass*.

Conclusion

Having discovered different types of TSR techniques (Greedy-based, Clustering-based, Search-based, Hybrid), as discussed by Khan[74], now it can be decided which kind will be applied in the optimisation method. As the main focus lies on the actual reduction of the test suite, every one of the techniques is applicable to the overall optimisation method. The clustering-based approach facilitates an algorithm for cluster analysis, employing two measures (*DisThreshold*, *SimThreshold*) to generate the clusters and a sampling algorithm to determine which test cases to use from the previously created clusters. The search-based approach calculates various potential solutions and applies an algorithm to find the best fitting, minimal reduced test set, using cost- and efficiency criteria. The hybrid approach consists of a combination of two or more of the discussed approaches.

3.3.2 Test Automation

Automated Software Testing (AST) is the automation of software tests, that would otherwise be executed manually.

Dustin et al.[24] discuss the differences between manual testing and automated testing being that AST:

- Enhances manual testing efforts by focusing on automating tests that manual testing can hardly accomplish.
- Is software development.
- Does not replace the need for manual testers' analytical skills, test strategy know-how, and understanding of testing techniques. This manual tester expertise serves as the blueprint for AST.
- Canot be clearly separated from manual testing; instead, both AST and manual testing are intertwined and complement each other.

Dustin's definition is targeted towards functional software testing, however, also applies to automated security testing, as there are tasks that a tester would not be able to do manually, e.g. sending corrupted network packets, and require a certain degree of software development/scripting in order to be accomplished. An automated security test cannot replace a tester's manual effort, as comprehensive security tests include source code reviews and other exploratory methods to find vulnerabilities.

As Meudec[56] describes, there are three different main categories of test automation:

- automation of administrative tasks, e.g. recording of test specifications and outcomes (useful for regression testing), test report generation;
- automation of mechanical tasks, e.g. the running and monitoring (for testing coverage analysis purposes) of the software under test within a given environment, capture/replay facilities allowing the automation of test suite execution;
- automation of test generation tasks, i.e. the selection and actual generation of test inputs

In this work, only the second item, the automation of test case running, will be discussed.

Test automation approaches have existed at least since 1999, as Archie states[6]. While in the beginning it was used primarily for test execution, nowadays automation efforts span from test case design to defect reporting; however, automation practices other than test execution still seem immature, as Garousi states[32]. According to Collins et al.[17], test automation has evolved to the key component in agile development and the core to agile testing.

Considering the future of test automation, Wiklund[95] argues, that ‘While explicit design of test cases surely will continue to have its place and serve a purpose, we believe that the future of automation must and will move towards a greater use of completely automatic testing: specify the goals of the test activity and the automation will fulfill the goals. Only corner cases that cannot be automatically reached will need manual intervention. Hence, the future of automation must consist of complete automation of the entire test process, not only the execution phase.’ They further state, that ‘they are highly complex and likely require skills in completely other areas than testing and developing the system under test’, making it infeasible to develop and maintain these future test automation approaches in the same teams as they are used in the software development process. They continue to mention the need for external development of test automation and provide the following reasons:

- It might be cheaper to externalise test automation. Small organisations often do not have the resources for test automation on such level and larger organisations might discover that centralising test automation in-house is cheaper and more efficient.
- The knowledge and skills required for specialised testing systems might be very special and difficult, rendering it infeasible to distribute the necessary knowledge to every development team.
- It might lead to better quality and maintainability of the testing infrastructure, if it is developed as a separate product by an external entity.

However, this also includes certain challenges[95]:

- *Implementation and Support Outside the Fail Fast-Fix Fast Loop*: the Fail Fast-Fix Fast Loop is the loop of discovering failures in the SUT as quickly as possible to be able to fix them as quickly as possible. Using external test automation development leads to a certain gap between testing and development. The right testing functionality is needed at the right time, including the knowledge of knowing how to apply it at the right time as well. If it fails to do so, additional cost can emerge and the test automation’s efficiency decreases.
- *Fear of Change*: it might be hard to replace parts of a test automation system with new solutions, and not the least because of psychological factors like employee’s fear of the effects of a new efficiency-increasing automation system and of change itself. Wiklund makes the case for change management as an important factor to properly manage the risks of employee’s fear of improvements in the testing procedure.

Xu et al.[98] present an approach and use formal threat modeling techniques and algorithms to automatically generate and execute security test cases. They use further constrained and formalised Petri nets to model threat nets and attack paths. These attack paths are then translated to security test cases in the C language or the Selenium browser automation framework[82]. However, as Xu et al. mention, their findings might be limited to systems similar to the ones used in the study.

When it comes to automated security testing, there are a few tools that can execute a set of standardised security test cases, which help to improve testing efficiency. The most widely used solution in this field is Nessus[91], which was first released in 1998 and developed to the most popular vulnerability scanning tool in use today[51]. The testing objects that can be covered with Nessus include[51]:

- **Network devices**: These include routers, firewalls, and printers
- **Virtual hosts**: These include VMware ESX, ESXi, vSphere, and vCenter
- **Operating systems**: These include Windows, Mac, Linux, Solaris, BSD, Cisco iOS, and IBM iSeries
- **Databases**: These include Oracle, MS SQL Server, MySQL, DB2, Informix/DRDA, and PostgreSQL
- **Web applications**: These include web servers and web services

Automated security testing tools for common vulnerabilities, like Nessus, are a good way of automatically covering weaknesses in a system's security. However, they can not replace vulnerability discovery for weaknesses specific to the SUT. It can only be a supplement for in-depth security analysis, as they only scan for known issues, which they look up in a database. Yet unknown vulnerabilities could always occur.

If automation of test cases is not planned or conducted properly, a risk materialises that the intended cost saving by test automation is minimised, or even results in increased costs, as Amannejad et al.[5], state. They further propose an approach to decide which test cases to automate and which not, in order to maximise Return of Investment (ROI). ROI they define as:

$$\frac{\Delta Benefit(automation over manual)}{\Delta Cost(automation over manual)} \tag{3.20}$$

Benefit is the sum of benefit factors, which need to be chosen depending on the context and objectives. An example might be the number of covered requirements. The same is applicable for cost. This might be simply money spent.

The basis for the approach is a Test Automation Decision Matrix (TADM), in which each row is a use case and the columns represent four areas in which test automation would be possible (Test Design, Test Scripting, Test Execution and Test Evaluation). Each cell is a binary value (0 or 1), indicating whether for the use case the given test phase should be automated or not. The TADM is then used in a genetic algorithm. The initial population is randomly-generated. In each iteration of the genetic algorithm, parents are selected using the fitness function - in this case the calculation of the ROI. A crossover operation selects a random position in the list of use cases in the TADM and creates two new solutions with the rows replaced by the other parent, starting from the chosen position. To avoid the algorithm getting stuck in local optima, a defined number of mutations are done in the next generation of solutions. After a defined number of iterations, the algorithm stops and the solution with the highest ROI is selected[5].

As stated earlier in this chapter, only the automation of test execution will be discussed in this chapter, rendering the TADMs a simple vector. The problem is now reduced to the decisions if a given use case's or test case's execution shall be automated or not. Given the 1-to-n constraint from section 3.4, that the test cases should only check one requirement and therefore not depend upon, or influence each other, the problem devolves to a problem of finding the local optimum. This means checking whether the cost of automating a test case and executing it automatically is higher or lower than the cost of doing so manually.

For the optimisation method, every test case, which fulfils Equation 3.21 shall be automated to reduce cost.

$$CostAutomation + CostAutomaticExecution * n < CostManualExecution * n \tag{3.21}$$

with n being the minimum expected number of times the test case is executed,

$CostAutomaticExecution$ being the cost of one automatic execution of the test case,

$CostManualExecution$ being the cost of one manual execution of the test case,

$CostAutomation$ being the cost of the initial automation of the test case.

If the cost of initial automation of the test case plus the cost of the actual automatic executions of the test case are lower than the cost of manual execution, then test automation is beneficial. It could be the case that automation pays off only after a certain amount of test executions, as shown by Figure 15:

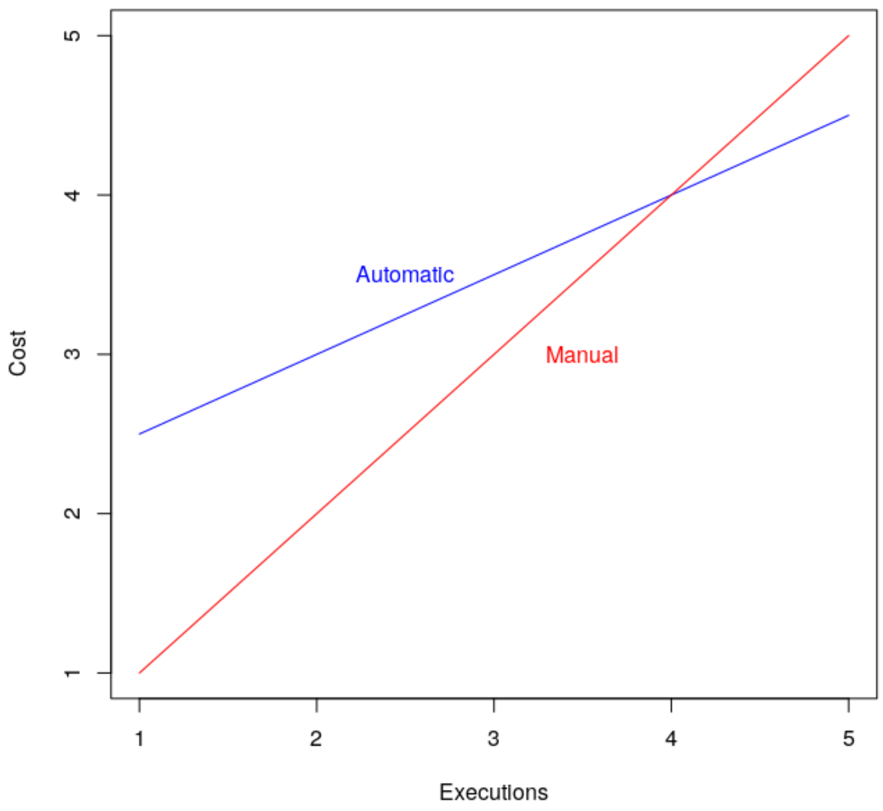

Figure 15: Cost of automatic vs. manual test execution

Test automation is done for every test case for which it would be reducing test execution cost.

3.3.3 Test Case Prioritisation

Ledru et al.[52] define test case prioritisation as follows: 'Test case prioritisation aims at finding an ordering which enhances a certain property of an ordered test suite'.

Wong et al. [97] have a more specific definition, taking into account cost of test case execution: 'A test set prioritization procedure sorts test cases in order of increasing cost per additional coverage, and then selects the top n test cases for revalidation.'

The test case prioritisation problem is formally defined by Rothermel et al.[77] as follows: 'Given: T, a test suite, PT, the set of permutations of T and f, a function from PT to the real numbers.
Problem: Find $T' \in PT$ such that $(\forall T'')$ $(T'' \in PT)$ $(T'' \neq T')$ $[f(T') \geq f(T'')]$'

The issue is to find a permutation of test cases that is considered 'best', using a function f, which evaluates and assigns a real number to every permutation. The defining element is the evaluation function, that calculates the score based on certain criteria that depend on the testing objectives. For this work in the security testing context, the criteria correspond to Section 2.1.4 about risk management:

1. Likelihood of occurrence of the vulnerability, which the system is tested for.
2. Cost of damage, that a possible exploitation of the vulnerability would cause.

These measures from the field of risk management can help to define a priority for the test cases.

Different approaches shall be presented and evaluated towards their fitness to the discussed criteria.

Ledru et al.[52] propose a test case prioritisation algorithm facilitating string distances. Every test case is represented lexicographically (as a text string), but does not take into account the test case's semantics. Such a string representation might be, e.g. a method's name. Using this string representation, a distance measure dd can be used to calculate the minimum distance between a test case t and an other test cases in a given test suite T':

$$dd(t, T') = min\{d(t, t_i) \mid t_i \in T' \; and \; t_i \neq dt\} \tag{3.22}$$

Then, a fitness function is defined as the sum of distances between each test case and the set of preceding test cases in P, a permutation of test cases of a test suite T:

$$\begin{aligned} &f(P) = \sum_{i=1}^{n} dd(t_i, T_{i-1}) \; where \\ &T_{i-1} = T \; for \; i = 1 \; and \\ &T_{i-1} = t_1, ..., t_{i-1} \; for \; i \geq 2 \end{aligned} \tag{3.23}$$

Ledru et al. then continues to elaborate the algorithm to prioritise the test cases[52]:

Given a test suite T, the following algorithm computes a prioritisation as a sequence P.

Algorithm 3.1: Ledru's test case prioritisation algorithm using string distance

```
1 Compute the distances for each pair of test cases in $T$;
2 Remove duplicates from $T$;
3 Find an element $t \in T$ with the maximum distance
    $dd(t, T)$, $T := T \setminus \{t\}$, $P := t$;
4 while $T$ is not empty do
5     Find an element $t \in T$ with the maximum distance $dd(t, P)$,
        $T := T \setminus \{t\}$, $P := P.t$ ($t$ is appended to the sequence);
6 end
7 Append duplicates to $P$;
8 return P;
```

Algorithm 3.1's complexity is $\mathcal{O}(n^2)$, with n being the test suite's size. It picks a test case and then adds the test case with the biggest distance from the already selected test cases. Adding very different test cases follows the approach that, if something very different is tested, the test is likelier to find different software failures and test coverage is increased. The most important thing in this approach is to find string representations of test cases that are precise enough to calculate a useful distance with the given distance measure. In combination with the string representation, the choice of distance measure also plays a crucial role. If one test case tests for buffer-overflow vulnerability in function A, it might be represented by 'test_buffer_overflow_A' and another test case, which tests the same in function B as 'test_buffer_overflow_B'. Using the Hamming distance approach, the distance would only be one, as only one character differs. The test case would most likely not be chosen in Ledru's algorithm, although it might have uncovered a serious buffer overflow vulnerability in function B. Another issue is that Ledru's approach does not consider the economic extent of possible failures, thus giving every failure the same weight in the prioritisation algorithm.

Khalilian et al. propose an approach[29] and an enhanced approach[49] to test case prioritization using historical data of previous executions of the test cases, to determine a new priority ranking. The historical test data is used in three ways:

1. Execution history, indicating how often a test case has been executed. Test cases not executed in the last sessions get higher priority to make sure they get executed at all.
2. The test case priority in previous testing sessions.
3. The relative effectiveness of a test case: the ratio of the number fault detection to the number of times a test case has been executed.

Khalilian [49] uses a variable coefficient approach, where the coefficients vary depending on how often the test case has been executed and if the test case revealed a fault. Using this approach, Khalilian managed to outperform another history based approach by Kim and Porter [50]. However, an obvious downside to history-based approaches remains: historical test execution data is needed to derive prioritisation metrics, which might not always be the case in the field of security testing.

An approach which corresponds to Section 2.1.4 about risk management includes that for every test case there shall be a score calculated, which results in a priority ranking of the test cases:

$$PriorityScore = Likelihood * Impact \tag{3.24}$$

The priority score is the product of likelihood of occurrence of the vulnerability, which the system is tested for, and severity of damage/impact, that a possible exploitation of the vulnerability would cause.

In the Common Criteria guidelines[19], a method for vulnerability assessment is presented to calculate the risk potential. The following factors are considered:

Elapsed Time means the amount of time between an attacker's identification of a vulnerability in the SUT and its successful exploitation.

Expertise is the level of knowledge an attacker must have in order to exploit the vulnerability. This may be in different fields like underlying operating systems, network protocols, exploitation techniques, et cetera. The following levels of expertise are distinguished:

- Laymen have no specific knowledge.
- Proficient persons are familiar with the general security behaviour of the system type.
- Experts are knowledgeable about the system's algorithms, protocols, structure, principles, concepts, security behaviour and about the tools and techniques necessary to perform attacks on the system type.
- Multiple Experts are people with expert level on multiple fields required to perform an attack on the system type.

Knowledge of Target of Evaluation (TOE) specifies how much knowledge about the SUT itself is necessary to exploit a vulnerability. In contrast, the previous factor *Expertise* refers to general knowledge in the fields. The different levels of knowledge are classified by the Common Criteria as follows[19]:

- Public information concerning the TOE (e.g. as gained from the Internet);

- Restricted information concerning the TOE (e.g. knowledge that is controlled within the developer organisation and shared with other organisations under a non-disclosure agreement),
- Sensitive information about the TOE (e.g. knowledge that is shared between discreet teams within the developer organisation, access to which is constrained only to members of the specified teams);
- Critical information about the TOE (e.g. knowledge that is known by only a few individuals, access to which is very tightly controlled on a strict need to know basis and individual undertaking).

The *Window of Opportunity* indicates how much access to the SUT is necessary to successfully exploit the vulnerability. This might be a certain amount of time or a number of data samples to collect. It is also possible that an attack might take a long time in preparation and there may be only a very small window of opportunity. The classification includes:

- Unnecessary/unlimited access means that no opportunity is necessary, as there is unlimited access to the SUT and there is no risk of being detected.
- Easy access involves access to the SUT for less than a day.
- Moderate access means access for less than a month.
- Difficult access includes required access for at least a month.
- The category None means, that it is not possible to obtain a window of opportunity large enough to perform the attack.

Equipment indicates which equipment is necessary to exploit a security vulnerability. It is classified as follows:

- Standard equipment is available to the attacker.
- Specialised equipment is equipment that is not available, but can be acquired without too much effort.
- Bespoke equipment is specifically produced for the attacker not available to the public.
- Multiple Bespoke includes the need for bespoke equipment of different kinds for the attack.

Table 7 shows an exemplary rating of the factors for vulnerability assessment[19]. The two asterisks (**) at the category None of the Window of Opportunity indicates that this can not be scored in the usual way, as the attacker has no opportunity to exploit the vulnerability in this case. It is not further specified how to calculate with this category.

Table 7: Exemplary rating of the factors for vulnerability assessment after [19].

Factor	**Value**
Elapsed Time	
<= one day	0
<= one week	1
<= two weeks	2
<= one month	4
<= two months	7
<= three months	10
<= four months	13
<= five months	15
<= six months	17
> six months	19
Expertise	
Layman	0
Proficient	3
Expert	6
Multiple experts	8
Knowledge of TOE	
Public	0
Restricted	3
Sensitive	7
Critical	11
Window of Opportunity	
Unnecessary / unlimited access	0
Easy	1
Moderate	4
Difficult	10
None	**
Equipment	
Standard	0
Specialised	4
Bespoke	7
Multiple bespoke	9

The sum of the classifications of the discussed categories can then be used to assess a vulnerability's severity. The lower the score, the less difficult it is to exploit a vulnerability.

The Open Web Application Security Project (OWASP)[66] suggests another method including various factors for estimating likelihood of occurrence and impact severeness. Each factor has different options, corresponding to a rating from 0–9. The overall rating of impact and likelihood is then represented by the mean of the ratings. The different factors are as follows (wording directly used from [66]):

Factors for Estimating Likelihood

The first set of factors are related to the threat agent involved. The goal is to estimate the likelihood of a successful attack by this group of threat agents. Use the worst-case threat agent.

- *Skill level*: How technically skilled is this group of threat agents? No technical skills (1), some technical skills (3), advanced computer user (5), network and programming skills (6), security penetration skills (9)
- *Motive*: How motivated is this group of threat agents to find and exploit this vulnerability? Low or no reward (1), possible reward (4), high reward (9)
- *Opportunity*: What resources and opportunities are required for this group of threat agents to find and exploit this vulnerability? Full access or expensive resources required (0), special access or resources required (4), some access or resources required (7), no access or resources required (9)
- *Size*: How large is this group of threat agents? Developers (2), system administrators (2), intranet users (4), partners (5), authenticated users (6), anonymous Internet users (9)

The next set of factors are related to the vulnerability involved. The goal here is to estimate the likelihood of the particular vulnerability involved being discovered and exploited. Assume the threat agent selected above.

- *Ease of discovery*: How easy is it for this group of threat agents to discover this vulnerability? Practically impossible (1), difficult (3), easy (7), automated tools available (9)
- *Ease of exploit*: How easy is it for this group of threat agents to actually exploit this vulnerability? Theoretical (1), difficult (3), easy (5), automated tools available (9)
- *Awareness*: How well known is this vulnerability to this group of threat agents? Unknown (1), hidden (4), obvious (6), public knowledge (9)
- *Intrusion detection*: How likely is an exploit to be detected? Active detection in application (1), logged and reviewed (3), logged without review (8), not logged (9)

Factors for Estimating Impact

Technical impact can be broken down into factors aligned with the traditional security areas of concern: confidentiality, integrity, availability, and accountability (see 2.1.1). The goal is to estimate the magnitude of the impact on the system if the vulnerability were to be exploited.

- *Loss of confidentiality*: How much data could be disclosed and how sensitive is it? Minimal non-sensitive data disclosed (2), minimal critical data disclosed (6), extensive non-sensitive data disclosed (6), extensive critical data disclosed (7), all data disclosed (9)
- *Loss of integrity*: How much data could be corrupted and how damaged is it? Minimal slightly corrupt data (1), minimal seriously corrupt data (3), extensive slightly corrupt data (5), extensive seriously corrupt data (7), all data totally corrupt (9)
- *Loss of availability*: How much service could be lost and how vital is it? Minimal secondary services interrupted (1), minimal primary services interrupted (5), extensive secondary services interrupted (5), extensive primary services interrupted (7), all services completely lost (9)
- *Loss of accountability*: Are the threat agents' actions traceable to an individual? Fully traceable (1), possibly traceable (7), completely anonymous (9)
- *Business Impact Factors*: The business impact stems from the technical impact, but requires a deep understanding of what is important to the company running the application. In general, you should be aiming to support your risks with business impact, particularly if your audience is executive level. The business risk is what justifies investment in fixing security problems.

Many companies have an asset classification guide and/or a business impact reference to help formalize what is important to their business. These standards can help to focus on what is truly important for security. If these are not available, then it is necessary to talk with people who understand the business to get their take on what is important.

The factors below are common areas for many businesses, but this area is even more unique to a company than the factors related to threat agent, vulnerability, and technical impact.

- *Financial damage*: How much financial damage will result from an exploit? Less than the cost to fix the vulnerability (1), minor effect on annual profit (3), significant effect on annual profit (7), bankruptcy (9)

- *Reputation damage*: Would an exploit result in reputation damage that would harm the business? Minimal damage (1), Loss of major accounts (4), loss of goodwill (5), brand damage (9)
- *Non-compliance*: How much exposure does non-compliance introduce? Minor violation (2), clear violation (5), high profile violation (7)
- *Privacy violation*: How much personally identifiable information could be disclosed? One individual (3), hundreds of people (5), thousands of people (7), millions of people (9)

For every test case tc of the test suite TS of size nts, the mean of the likelihood factors LF of size nlf and the mean of the impact factors IF of size nif are calculated and applied to Equation 3.24:

$$PriorityScore(tc_i) = \frac{\sum_{j=1}^{nlf} lf_j * \sum_{k=1}^{nif} if_k}{nlf * nif} \tag{3.25}$$

A possible variant of this priority score is a test costs weighted priority score, in which the priority score is weighted by the reciprocal test costs $tcost$ of the given test case:

$$CostWeightedPriorityScore(tc_i) = \frac{\sum_{j=1}^{nlf} lf_j * \sum_{k=1}^{nif} if_k}{nlf * nif * tcost_i} \tag{3.26}$$

This division by the test costs results in a measure which indicates how much 'gain of priority score' is obtained per unit of test cost, for every test case.

For the optimisation method, the priority score, as formulated in Equation 3.25, will be chosen as the method's priority measure for test case prioritisation.

3.4 The Security Test Concept Optimisation Method

Chapters 3.3.1, 3.3.2 and 3.3.3 presented various methods in the fields of test suite reduction, test automation and test case prioritisation, out of which the security test optimisation method will be created.

The input is the original test suite, and the output is an optimised test suite. Figure 16 shows a conceptual diagram of all the steps of the optimisation method. The test suite reduction step is executed first, as there is no necessity to evaluate test cases for automation or calculate a priority, if they are removed anyway. Afterwards in the test automation step, every test case is evaluated if it should be automated, resulting in different assigned cost. These cost are then used in the test case prioritisation step, where a priority is assigned to every test case and a budgetary constraint applied.

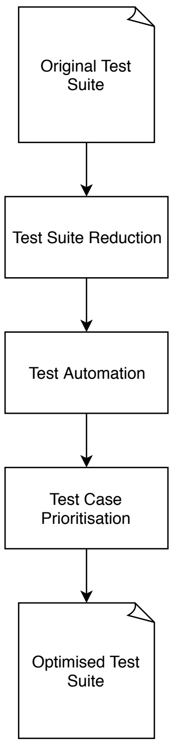

Figure 16: Conceptual diagram of the optimisation method.

Test Suite Reduction

A constraint shall be applied to the given test suite for test suite reduction: The test suite shall be designed, so that every test case satisfies only one requirement, however, it is possible that a requirement is satisfied by more than one test case. This constraint shall be referenced as ‘1-to-n constraint’ or as ‘normalisation’.

Although, it is possible to use other approaches, for the application of the optimisation method, it is suggested to facilitate a greedy approach, as it is quite easy and intuitive to apply and performs well, even with larger test suite sizes. As stated by Chen[16], the greedy approach has a worst case time-complexity of $\mathcal{O}(min(m,n)nk)$, with m being the number of requirements, n being the number of test cases and k being the maximum number of requirements satisfied by a single test case. Given the 1-to-n constraint, the greedy TSR algorithm changes to a simple 1-to-n redundant test case removal problem[16]. Thus, the worst case time-complexity changes to $\mathcal{O}(min(m,n)n)$, as k is 1.

According to Rothermel[78], test suite reduction can lead to significant losses in fault-detection effectiveness. If the requirement is the check for a specified fault, like it is the case with security tests, a loss in fault-detection effectiveness in the greedy algorithm can only be possible, if a test case tested for more than the one requirement it is assigned to. This, however, is excluded by the 1-to-n constraint. Logically this means that in this case there can be no loss in fault-detection effectiveness.

The following steps will reduce the test set using a greedy algorithm.

1. According to the 1-to-n constraint from section 3.3.1, make sure every test case tests exactly one requirement. A requirement may be covered by multiple test cases. If a test case covers multiple requirements, split the test case into multiple test cases, so that every test case tests only one requirement.
2. For every test case $tc_i \in TS$ covering a requirement $r_j \in R$ check if there is another test case $tc_k \in TS$ which covers the same requirement $r_j \in R$. If one or multiple of such a test cases are found, remove them: $TS = TS \setminus tc_k$.

The resulting test suite RS is then used for the test automation step.

Test Automation

The reduced test suite TS is analysed for feasibility of test automation:

1. For every test case $tc_i \in TS$ check:

$$CostAutomation + CostAutomaticExecution * n < CostManualExecution * n \tag{3.27}$$

   with n being the minimum expected number of times the test case is executed,

   $CostAutomaticExecution$ being the cost of one automatic execution of the test case,

   $CostManualExecution$ being the cost of one manual execution of the test case,

   $CostAutomation$ being the cost of the initial automation of the test case.

   If the cost of initial automation of the test case plus the cost of the actual automatic executions of the test case are lower than the cost of manual execution, then test automation is beneficial and the test case will be automated.

Test Case Prioritisation

The resulting test cases are now prioritised to determine which test cases to execute first. A budgetary limit will then be applied and it will therefore be decided which test cases are not executed at all. Some test cases are essential for the security test as a whole and cannot be removed from the test suite to execute. For these test cases, an analysis for why they are so important can be obtained and the scores assigned accordingly to make sure these test cases are in the test suite.

1. Every test case $tc_i \in TS$ must be assigned a cost value to determine how expensive it is to execute the test case, including possible test automation cost determined in the previous step.

2. Every test case $tc_i \in TS$ must be assigned an impact value to determine the impact of the tested requirement (the impact of vulnerability exploitation). The OWASP Risk Rating Methodology[66] can be used for this, as described in Section 3.3.3.

3. Every test case $tc_i \in TS$ must be assigned a likelihood value to determine how likely the exploitation of the vulnerability is. The OWASP Risk Rating Methodology[66] can be used for this.

4. Every test case $tc_i \in TS$ must be assigned a prioritisation score according to Equation 3.25:
$$PriorityScore(tc_i) = \frac{\sum_{j=1}^{nlf} lf_j * \sum_{k=1}^{nif} if_k}{nlf * nif}$$

5. Determine a budget for security test execution. The executed test cases must not exceed this cost limit.

6. Select test cases for execution according to prioritisation. If the selection of the next test case would violate the cost limit, search for the next test case according to prioritisation ranking that does not violate the cost limit, until there are only test cases left that would violate the limit, or no test case is left at all.

The result is a reduced and automated test set with a given execution ordering according to the test case priority.

CHAPTER 4

Testing the Method on a Decentralised Component with High Protection Need

After establishing a security test optimisation method, consisting of steps for test suite reduction, test automation and test case prioritisation, this methodology will be tested on a real world example in the eHealth field. Section 4.1 describes the system under test, a component used in the eHealth field. The criteria used for comparison of the test suites before and after the optimisation method's application are discussed in Section 4.2. In Section 4.3 the optimisation method is applied on the original test suite for the SUT. Finally, in Section 4.4 the results of the method's application are discussed.

4.1 The System Under Test

The System Under Test is a hardware component with a Linux operating system running various services, used in the eHealth field. These services are used to establish a connection to a centralised network, handle sensitive, personal health data and are further described below Figure 17. The component is part of a hardware appliance used to connect a doctor's office to an eHealth infrastructure, which offers eHealth services to the doctor and the patient.

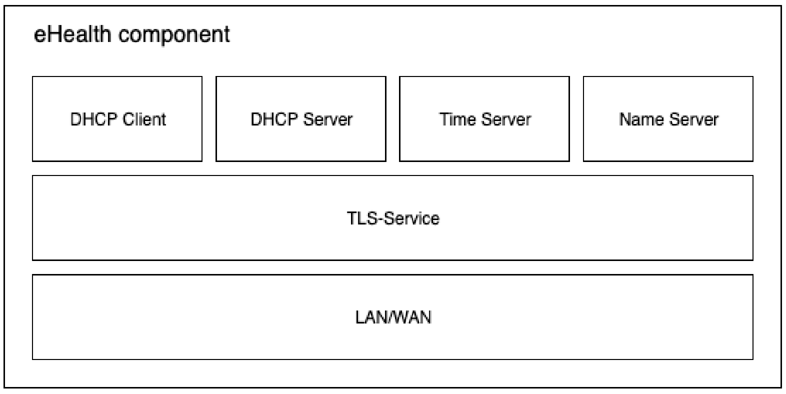

Figure 17: The eHealth component including its subcomponents.

Figure 17 shows a schematic of the discussed component, including its subcomponents. These consist of:

- LAN/WAN: this subcomponent handles the connection of the component itself to the network. LAN for internal communication, WAN for connection to the outside world.
- TLS Service: the TLS service offers encryption and decryption services and centralises cryptographic operations. In the test suite, the test cases are classified into 'TLS Service Incoming', for connections from a client to the TLS service and 'TLS Service Outgoing' for connections going out from the TLS service.
- DHCP Client: the DHCP client queries an external DHCP server for IP addresses on the WAN interface.
- DHCP Server: the DHCP server manages IP addresses of clients in the LAN.
- Time Server: matters of managing correct date and time and synchronisation thereof are handled by the time server.
- Name Server: the name server uses Domain Name System (DNS) for mapping between host names and IP addresses.

For each of the subcomponents, several security tests are presented in Appendix A.2. This test suite shall be the basis for the optimisation method.

4.2 Criteria

Criteria are the necessary numbers to indicate, if relevant changes took place. By before/after comparison, these changes can be detected and quantified, allowing further analysis of the meaning and consequences of the differences in numbers. If, for example, it seems to be a very hot summer, a person might make a subjective observation: ‘It gets hotter every summer!’. This observation might or might not be true. In order to verify the statement, it is necessary to apply a criterion: the difference in mean temperature in different summers. Only then can the statement be supported or falsified and an objective observation made.

In order to evaluate the optimisation method, several criteria have been established for comparison of the test suites before and after the optimisation method’s application. The following criteria will be facilitated in comparison:

- *Number of test cases reduced*: the number of test cases that have been reduced with the test suite reduction step. The more redundant test cases could be removed, the better, as the reduction indicates less effort without the loss of effectiveness and therefore increased efficiency. For the optimisation method, reduced test cases result in a cheaper test execution with the same coverage. The number of reduced test cases, however, also depends on the original test suite. If there are no redundant test cases in the original test suite, then no test could be reduced, rendering this number 0. This would mean, that the original test suite was not suitable for the purpose of this work: showing if the method can optimise test suites. Given a number of 0, it is impossible to say if the original test suite did not contain any redundant test cases, or if the method did not work. If that would be the case, the application would need to be executed on a different test suite.

- *Difference in number of unique test cases*: the number of unique test cases that are chosen for execution before and after the application of the optimisation method. Through test case reduction, this number should rise. Given a defined budgetary constraint, with the reduction of redundant test cases, more unique test cases should be covered with that same budget. This indicates increased testing coverage without increased effort and therefore boosts the likelihood of finding security vulnerabilities. The higher the number, the bigger the increase in effectiveness, as the coverage was higher. This increase in coverage could be achieved using the same amount of resources, which indicates also a boost in efficiency. The difference in number of unique test cases is closely related to the number of test cases reduced. More reduced test cases mean more space in the budgetary constraints for other unique test cases. However, as there are different test cases being added, than being reduced, the difference in number of unique test cases has not a 1:1 correlation to the number of test cases reduced. This is further true, as test case automation might reduce the cost of test cases.

- *Averted potential damage*: the potentially averted damage cost points out how much potential damage cost the vulnerabilities, that have been tested for, would have caused. In the test case prioritisation step the cost of potential damage is calculated for every test case, as described in Section 4.4.3. For comparison, the sum is calculated for the original test suite and the resulting optimised test suite. The difference can be seen as a measure for effectiveness of the test suites. A higher number means, that more damage could be averted by testing for more security issues, using the same budget.

On the way to finding the three criteria, a few other criteria have been analysed and rejected as insufficient. The total cost of the test suite before and after the application would be an unsuitable criterion as the total cost depends on the budgetary constraint and can be chosen arbitrarily. Using the same budget before and after the optimisation method's execution would simply result in the number 0 for this criterion. Any change in budget would result in a number reflecting exactly that arbitrary change. Also, neither the average cost of test cases could be used, as the cost does not necessarily reflect the effectiveness of the test case. There might be many low-costing test cases or few high-costing test cases, which would result in significant differences in the average cost of test cases, but say nothing about the test suite's effectiveness. Finally, the absolute number of test cases in the test suite could also not be used as an indicator for test suite efficiency or effectiveness, for the same reasons the average cost of test cases could not be used. The cost or quantity of test cases in the test suite do not reflect the test suite's effectiveness of efficiency.

4.3 Application of Optimisation Method

The original test suite was obtained by analysing the SUT, its interfaces (ingoing and outgoing, external and internal) and where vulnerabilities could occur in those interfaces in various layers of interaction (e.g. network layer). This means, that for every interface the whole stack of underlying protocols has been analysed and test cases for potential vulnerabilities have been created. Further, the functionality lying behind the interface was also analysed for possible security issues, thus covering a large part of the attack surface for someone trying to attack that specific interface.

4.3.1 Test Suite Reduction

Before the TSR approach can be executed, the first step of the optimisation method, according to Section 3.4, is to make sure that every test case tests only one requirement. This constraint, called '1-to-n constraint', is a necessary precondition for the test suite reduction approach, in order to avoid loss of coverage, as also described in Section 3.4. Every test case is analysed and checked, if it tests for more than one requirement. If this is the matter, the test case is split up, so that every test case only tests for one requirement. Applying this constraint to the original test suite from Appendix A.2 results in the replacements in the given categories in Table 8 and to the test suite described in Appendix A.4. Table 8 shows the IDs of the replaced test cases and the IDs of the test cases which took their place, for each category. In the case of the 'Name Service' category, test cases with IDs 1 to 2 have been replaced by the test cases with IDs 9 to 14.

Category	**Replaced Test Cases**	**Replacement**
General	2–3	9–14
LAN/WAN	1–2	14–19
DHCP-Client	1–2	10–15
DHCP-Server	1–2	11–16
Name Service	1–2	9–14
TLS Service Incoming	1–2	43–48
TLS Service Outgoing	1–2	38–43
Time Service	1–2	16–21

Table 8: By normalisation replaced test cases.

The replacement of test cases with split test cases normalises the test suite. This normalisation process leads to a larger test suite, which might seem counterproductive at first thought. This, however is an essential step to avoid loss of coverage through the TSR step and also might increase the possibility that further redundant test cases can be reduced. If a test case tc_1 tests for requirements r_1 and r_2 and another test case tc_2 tests for requirements r_1 and r_3, then the normalisation step leads to four test cases:

- tc_a testing for r_1, created by splitting tc_1
- tc_b testing for r_2, created by splitting tc_1
- tc_c testing for r_1, created by splitting tc_2
- tc_d testing for r_3, created by splitting tc_2

The test suite reduction step would then reduce test case tc_c, as it tests for the same requirement as tc_a, without losing any coverage.

The next step consists of applying the greedy test suite reduction method, as described in Section 3.4, ensuring that every requirement is only tested by one test case. If a test case is found, that tests for a requirement, which is already covered by another test case, the found test case is removed from the test suite. This analysis is done for every test case and establishes a 1-to-1 relationship between a requirement and a test case and results in unique test cases. The reduction of a further test case would result in a loss of coverage. In comparison to the normalised test suite from Appendix A.4, many network specific tests could be removed, as the SUT's subcomponents share the same network stack. The reduced test cases can be found in Table 9.

Category	**Reduced Test Cases**
General	5–14
DHCP-Client	3–6, 10–15
DHCP-Server	3–6, 11–16
Name Service	3–6, 9–14
TLS Service Incoming	3–6, 43–48
TLS Service Outgoing	3–6, 38–43
Time Service	3–6, 16–21

Table 9: The test cases reduced by test suite reduction.

In the test suite reduction method, the test suite could be reduced by 70 test cases.

4.3.2 Test Automation

The security test optimisation method continues with the step ‘test automation’ from Section 3.4. For this step, it is necessary to determine the following parameters for every test case:

- *Cost of automation*: How much it costs to automate the execution of a test case, otherwise executed manually.
- *Cost of an automatic execution*: The cost of the execution of an already automated test case.
- *Cost of a manual execution*: The cost of a manual execution of a test case.

To determine, whether to automate a test case's execution, Equation 3.21 will be used with the determined parameters:

$$CostAutomation + CostAutomaticExecution * n < CostManualExecution * n$$

The number of executions for this test case, n, shall be 2 for this. These values need to be estimated, or deduced from historic data. In this case, the different cost variables will be estimated. Given the following estimations for a test case tc_1:

- Cost of automation: 450
- Cost of automatic execution: 10
- Cost of manual execution: 250

For $n = 2$, the condition to automate the test case would be $450 + 10 * 2 < 250 * 2$. As $470 < 500$ is true and the condition holds, the test case is selected for automation. The test cases which have been chosen for automation are indicated in Appendix A.5 and summarised in Table 10.

Category	**Automated Test Cases**
General	4
LAN/WAN	6, 10, 12, 18
DHCP-Client	8, 9
DHCP-Server	8, 9, 10
Name Service	7, 8, 16, 18
TLS Service Incoming	7, 12, 15–18, 20, 24–28, 34–36, 40
TLS Service Outgoing	7, 17, 20–24, 27–30, 35, 37
Time Service	12

Table 10: The test cases chosen for automation.

Of the reduced test suite, 45 test cases have been chosen for automation.

4.3.3 Test Case Prioritisation

The final step consists of prioritising the test cases for maximising the reduction of potential damage cost, as discussed in Section 3.4. The potential damage cost is the cost a vulnerability would cause when exploited, because there was no test for it. For every test case, the following parameters need to be estimated:

- Cost of execution (be it automated or manually executed)
- Impact score, as estimated in Section 3.3.3
- Likelihood score, as estimated in Section 3.3.3
- Priority score according to Equation 3.24

Further, a budget must be defined, which cannot be violated when executing the test suite. With the priority score assigned to every test case, an ordering of the test cases is now possible.

Appendix A.6 shows the reduced and partially automated test suite with the necessary values for prioritisation assigned, and the test cases sorted by priority score. In the final and optimised test suite, as seen in Appendix A.7 a budgetary constraint of EUR 20.000 has been applied, as described in Section 3.4, and 68 test cases have been selected.

This concludes the application of the optimisation method to the real world example and leads to the detailed analysis of the results.

4.4 Results

After the application of the optimisation method, the results will be discussed to determine if the optimisation method improved the test concept's efficiency. Therefore, the three evaluation criteria discussed in Section 4.2 shall be used: number of test cases reduced, difference in number of unique test cases and averted potential damage.

4.4.1 Number of Test Cases Reduced

In the test suite reduction step, 70 test cases (39% of the normalised test suite) could be removed, as they were redundant and did not add any value to the test suite. Figure 18 gives a visual representation of the amount of reduced test cases. Although the number might seem high and certainly shows that the test suite reduction algorithm is capable of reducing the test suite, this absolute number must be considered with caution. It is influenced by the quality of the initial test suite. If the original test suite includes a lot of redundant or partially redundant test cases (which might turn to fully redundant test cases after normalisation), this number will be higher. If it includes fewer redundancies, it will be lower. However, the reduction step helps to increase the test suite's efficiency as the same requirements are covered with less effort.

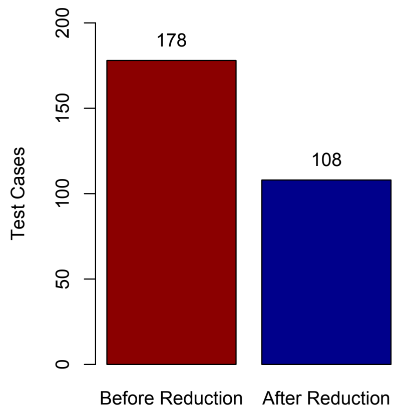

Figure 18: Number of test cases before and after test suite reduction.

4.4.2 Difference in Number of Unique Test Cases

The number of unique test cases, given a budgetary constraint of EUR 20.000, is as follows.

For the original test suite, the selection of test cases until the budget is reached was done as described in Section 3.3.3. This resulted in the selection of 31 test cases without any prioritisation, in the order of creation. Out of these 31 test cases, 18 were unique test cases. Given the monetary valuation as described in section A.6, this would amount to EUR 9.400 worth of redundantly executed test cases. Table 11 shows the selected test cases.

ID	Title	Category	Cost in EUR	Unique
1	Known Vulnerabilities	General	400	YES
2	ICMP Attacks	General	800	YES
3	IP Attacks	General	2700	YES
4	Nessus-Scan	General	200	YES
5	Open Ports	General	200	YES
6	Ping of Death	General	200	YES
7	SYN Flooding	General	200	YES
8	TCP RST	General	200	YES
1	ICMP Attacks	LAN/WAN	800	NO
2	IP Attacks	LAN/WAN	2700	NO
3	Open Ports	LAN/WAN	200	NO
4	Ping of Death	LAN/WAN	200	NO
5	SYN Flooding	LAN/WAN	200	NO
6	TCP RST	LAN/WAN	200	NO
7	ConfigureRoutes Input Validation	LAN/WAN	400	YES
8	InitLAN Input Validation	LAN/WAN	400	YES
9	InitWAN Input Validation	LAN/WAN	400	YES
10	SendDataExternal Change Gateway	LAN/WAN	1000	YES
11	SendDataExternal Input Validation	LAN/WAN	400	YES
12	SendDataHealth Client/BusinessModule Identity	LAN/WAN	1800	YES
13	SendDataHealth Input Validation	LAN/WAN	400	YES
1	ICMP Attacks	DHCP-Client	800	NO
2	IP Attacks	DHCP-Client	2700	NO
3	Open Ports	DHCP-Client	200	NO
4	Ping of Death	DHCP-Client	200	NO
5	SYN Flooding	DHCP-Client	200	NO
6	TCP RST	DHCP-Client	200	NO
7	Init Input GetInformation	DHCP-Client	400	YES
9	GetInformation DOS	DHCP-Client	350	YES
1	ICMP Attacks	DHCP-Server	800	NO
10	DHCP Init Stress Test	DHCP-Server	150	YES

Table 11: The test cases chosen from the original test suite, given a budget of EUR 20.000.

In the final test suite, as shown in Appendix A.7, 68 test cases have been selected, which, after the test suite reduction step, are unique per definition. False positives are per definition impossible, as only test cases are removed, which cover a requirement, that has already been covered by another test case. This is an increase in covered test cases by 50, which is more than 4.7 times the amount of selected test cases in the original test suite. Of course, this absolute result is also dependent on the original test suite's quality, nevertheless, it clearly shows the effectiveness of test suite reduction, prioritisation and also test automation, as it reduces cost over multiple iterations. Through increased coverage, more test cases can be executed, thus more security issues tested for, with the same resources. Without the step of test automation, only 24 test cases would have been covered, as shown in Table 12, which indicates a relevant impact of optimisation through automation on the overall result. Figure 19 shows the number of covered unique test cases in the original test concept, the reduced but not automated test concept, and the reduced and automated test concept.

ID	**Title**	**Category**	**Impact Value**
1	Known Vulnerabilities	General	5,875
4	Nessus-Scan	General	5,875
7	TLS Fuzzing	TLS Service Incoming	5
8	Overwrite Static Network Info	DHCP-Client	4,875
18	IP Spoofing Attack	LAN/WAN	4,75
8	DNS Spoofing	Name Service	4,5
12	Allowed Cipher Suites	TLS Service Outgoing	4,5
10	SSLv2 Usage	TLS Service Outgoing	4,5
11	TLSv1.	TLS Service Outgoing	4,5
39	Signature Algorithm Selection	TLS Service Incoming	4,5
13	Server Side Change of Cipher Suites	TLS Service Outgoing	4,5
21	Modified MAC	TLS Service Outgoing	4,375
24	Modified MAC	TLS Service Incoming	4,375
15	Broken Extensions	TLS Service Outgoing	4,25
8	DHCP Spoofing Attack	DHCP-Server	4,125
12	Atypical Padding	TLS Service Incoming	4,125
16	MD5 check	TLS Service Incoming	4,125
10	Allowed Ciphers	TLS Service Incoming	3,875
11	Forbidden Ciphers	TLS Service Incoming	3,875
8	TLS Version Compliance	TLS Service Outgoing	3,875
9	TLS Unknown Version Number	TLS Service Outgoing	3,875
8	TLS Version Checks	TLS Service Incoming	3,875
4	Ping of Death	LAN/WAN	3,625
9	TLS Default Version	TLS Service Incoming	3,625

Table 12: The selected test cases from the final test suite without automation.

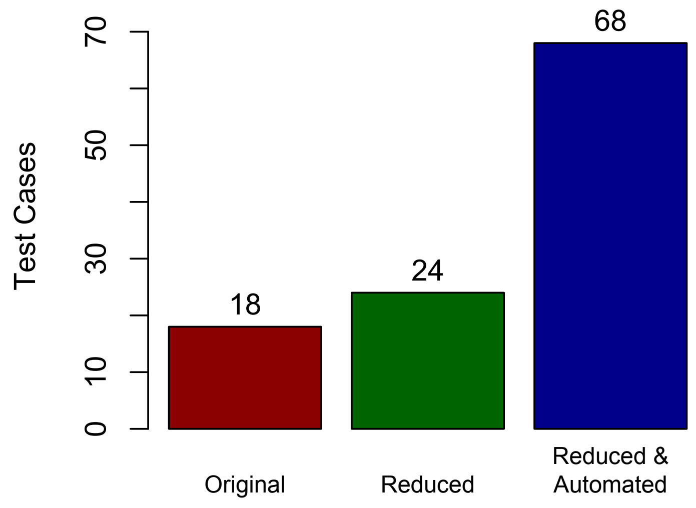

Figure 19: Number unique test cases in the original, reduced only, and reduced and automated test concept

4.4.3 Averted Potential Damage

The potential damage averted security issues would have caused is difficult to measure or estimate, as it can vary from context to context. In larger companies, for example, security issues in an application might prove much more expensive than in smaller ones. Therefore, the comparison of potential damage averted shall be undertaken not in EUR, but in the sum of the impact scores according to [66], of the unique test cases within the budget. Tables 13 and 14 show the selected test cases of the original test suite and the optimised test suite and their assigned impact values.

ID	Title	Category	Impact Value
1	Known Vulnerabilities	General	5,875
4	Nessus-Scan	General	5,875
3	IP Attacks	General	4,375
11	SendDataExternal Input Validation	LAN/WAN	4,125
10	SendDataExternal Change Gateway	LAN/WAN	3,875
7	ConfigureRoutes Input Validation	LAN/WAN	3,875
8	InitLAN Input Validation	LAN/WAN	3,625
6	Ping of Death	General	3,625
9	InitWAN Input Validation	LAN/WAN	3,625
12	SendDataHealth Client/BusinessModule Identity	LAN/WAN	3,5
13	SendDataHealth Input Validation	LAN/WAN	3,375
7	SYN Flooding	General	3,125
8	TCP RST	General	3,125
10	DHCP Init Stress Test	DHCP-Server	3,125
5	Open Ports	General	2,875
2	ICMP Attacks	General	2,75
7	Init Input GetInformation	DHCP-Client	2,5
9	GetInformation DOS	DHCP-Client	1,75

Table 13: The test cases chosen from the original test suite and their impact values.

ID	Title	Category	Impact Value
1	Known Vulnerabilities	General	5,875
4	Nessus-Scan	General	5,875
7	TLS Fuzzing	TLS Service Incoming	5
8	Overwrite Static Network Info	DHCP-Client	4,875
18	IP Spoofing Attack	LAN/WAN	4,75
8	DNS Spoofing	Name Service	4,5
12	Allowed Cipher Suites	TLS Service Outgoing	4,5
10	SSLv2 Usage	TLS Service Outgoing	4,5
11	TLSv1.	TLS Service Outgoing	4,5
39	Signature Algorithm Selection	TLS Service Incoming	4,5
13	Server Side Change of Cipher Suites	TLS Service Outgoing	4,5
21	Modified MAC	TLS Service Outgoing	4,375
24	Modified MAC	TLS Service Incoming	4,375
23	Modified Cipher Text	TLS Service Outgoing	4,375
37	Message Duplication	TLS Service Incoming	4,375
38	Message Skipping	TLS Service Incoming	4,375
15	Broken Extensions	TLS Service Outgoing	4,25
8	DHCP Spoofing Attack	DHCP-Server	4,125
12	Atypical Padding	TLS Service Incoming	4,125

16	MD5 check	TLS Service Incoming	4,125
28	Client Hello	TLS Service Incoming	4,125
21	ECDHE	TLS Service Incoming	4,125
25	Modified Finished Message	TLS Service Incoming	4,125
26	Modified Cipher Text	TLS Service Incoming	4,125
27	Modified Padding	TLS Service Incoming	4,125
31	Client Hello Truncating	TLS Service Incoming	4,125
42	Finished Message Truncating	TLS Service Incoming	4,125
14	Atypical Padding	TLS Service Outgoing	4,125
18	Hash and Signing Algorithms	TLS Service Outgoing	4,125
22	Modified Finished Message	TLS Service Outgoing	4,125
24	Modified Padding	TLS Service Outgoing	4,125
27	Invalid Handshake Messages	TLS Service Outgoing	4,125
28	Invalid Additional Messages	TLS Service Outgoing	4,125
29	Invalid Order of Messages	TLS Service Outgoing	4,125
31	Padded/Truncated Finished Message	TLS Service Outgoing	4,125
32	Padded/Truncated Hello Message	TLS Service Outgoing	4,125
33	Padded/Truncated KeyExchange Message	TLS Service Outgoing	4,125
36	ECDHE Cipher Support	TLS Service Outgoing	4,125
37	Bad ECDHE Bad Messages	TLS Service Outgoing	4,125
20	Invalid Content	TLS Service Outgoing	4,125
32	Session Renegotiation	TLS Service Incoming	4
8	TLS Version Checks	TLS Service Incoming	3,875
10	Allowed Ciphers	TLS Service Incoming	3,875
11	Forbidden Ciphers	TLS Service Incoming	3,875
8	TLS Version Compliance	TLS Service Outgoing	3,875
9	TLS Unknown Version Number	TLS Service Outgoing	3,875
35	Zero Length Data	TLS Service Incoming	3,875
29	Client Hello Record Overflow	TLS Service Incoming	3,875
33	Session Resumption	TLS Service Incoming	3,875
35	Invalid Session ID	TLS Service Outgoing	3,875
7	TLS Handshake	TLS Service Outgoing	3,875
20	Early invalid application data	TLS Service Incoming	3,75
13	Extensions	TLS Service Incoming	3,75
12	Data Manipulation	Time Service	3,75
4	Ping of Death	LAN/WAN	3,625
9	TLS Default Version	TLS Service Incoming	3,625
36	Invalid Session ID	TLS Service Incoming	3,625
9	DHCP Starvation Attack	DHCP-Server	3,625
19	Early application data	TLS Service Incoming	3,5
34	Invalid Compression	TLS Service Incoming	3,5
41	SSLv2 hello	TLS Service Incoming	3,5
19	Early Payload	TLS Service Outgoing	3,5

23	Empty Extension	TLS Service Incoming	3,5
7	DNS Amplification Attack	Name Service	3,375
5	SYN Flooding	LAN/WAN	3,125
6	TCP RST	LAN/WAN	3,125
10	DHCP Init Stress Test	DHCP-Server	3,125
3	Open Ports	LAN/WAN	2,875

Table 14: The test cases chosen from the optimised test suite and their impact values.

The comparison of Tables 13 and 14 here clearly depicts a significant difference in test suite size after test case prioritisation and application for the budgetary limit.

The sum of the impact values of the original test suite's selected test cases is 65 and the one of the optimised test suite's is 276. This is an increase by a factor of 4.25, indicating that 4.25 times the potential damage of the security issues, has been avoided. This indicates an increase in test suite effectiveness. As shown in table 12, when leaving out the test automation step, the value of potential damage prevention sinks to 105, which is 171 less than 276 (the value of averted potential damage of the final test suite), which points out the impact test automation can have on the prevention of potential damage through security testing. Figure 20 compares the original test suite, the reduced test suite, and the reduced and automated test suite, visually.

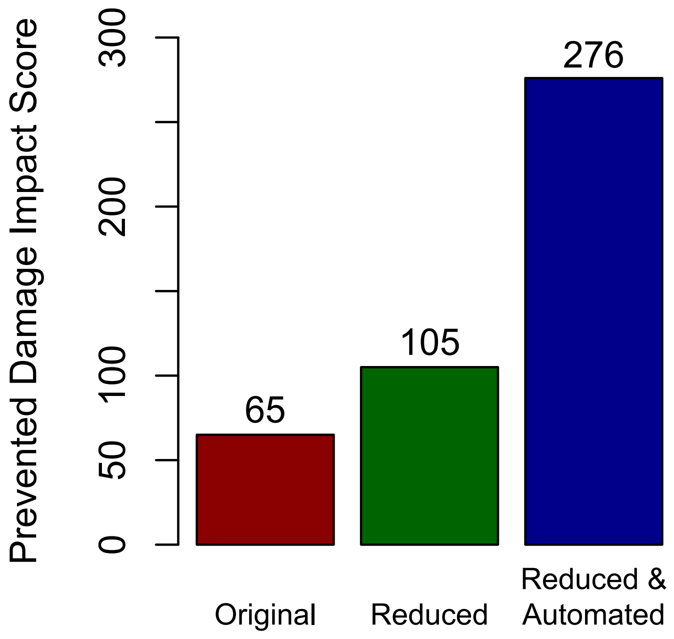

Figure 20: Prevented potential damage in impact scores according to [66]

CHAPTER 5

Conclusion & Outlook

When sensitive data is handled in an application, e.g. eHealth applications, which often process sensitive health data, security tests are a way to reduce the likelihood of security vulnerabilities. As Geer [34] mentions, it is impossible to prove a system's security, but only its insecurity. Proving its security would imply enumerating all possible vulnerabilities in all possible places and checking each one of them, which is impossible, as there are countless known vulnerabilities and possibly numerous yet unknown ones. For a security tester this means that a system could be tested forever, which is infeasible. Explorative testing without constraints or a security testing concept seems, therefore, threatening for a security test's efficiency and effectiveness. Therefore, to avoid getting lost, enough time must be spent on test conception and also for test concept optimisation, to manage the test's success in both efficiency and effectiveness.

The proposed optimisation method consists of three parts: test suite reduction, test case automation, and test case prioritisation.

Section 3.3.1 introduced the basics and various different techniques of test suite reduction. For the optimisation method a simple greedy algorithm was chosen, as it can be applied easily and therefore increases the likelihood of being used. For test case automation, an analysis is done for every test case, if the automation and the automated execution cost are lower than the manual execution cost. If that is the case, then the test case is being automated. Finally, the test case prioritisation step helps to decide the ordering of test case execution. The OWASP risk rating methodology[66] is used to score impact and likelihood for every vulnerability tested for and then calculate an overall priority score. The higher the score, the higher up the priority of the test case. The result is a prioritised and minimised test suite with the information if a test case shall be optimised.

When testing the optimisation method on a real-world example in the eHealth field, it showed a significant improvement in the chosen criteria described in Section 4.4: the number of test cases reduced, and the increase in coverage of unique test cases and averted potential damage. This increase shows that the optimisation method can be used to optimise a security test's efficiency and effectiveness. However, the absolute numbers must be handled with caution, as they highly depend on the rate of redundancy and the difference in ordering of the original test suite. This makes it impossible to make absolute statements about the optimisation method's quality.

Therefore, it could be subject to further study to conduct performance comparisons of this optimisation method to other techniques of test suite optimisation. At the time of writing, no other methodology which combines test suite reduction, test case automation, and test case prioritisation, is known to the author, for either security tests, nor for regular functional tests. Comparisons could be undertaken by looking at the three techniques separately and comparing the performance with their counterparts. These comparisons could include further metrics to analyse their performance in different lights and requirements.

The larger that SUTs get, the larger the security test concepts are to cover the system properly. This also increases the relevance to automate the optimisation method itself in order to avoid the excessive cost of test concept optimisation. The optimisation method's automation could be another subject for further research.

In a software engineering project, the optimisation method could also be integrated in the requirements engineering process to make sure traceability of security requirements is intact and every requirement is covered not only before but also after the optimisation method's execution. How a seamless integration into the requirements engineering process could look is subject to further study.

APPENDIX A

Appendix

Appendix A.1 shows the source code listing used in this thesis. Appendix displays the original test suite, which is the basis of the application of the optimisation method. The normalised test suite, after the application of the 1-to-n constraint, also called normalisation, is shown in Appendix . Appendix includes the reduced test suite, Appendix the partially automated one and Appendix holds the test suite after the prioritisation step. Finally, Appendix shows the final test suite after the application of the optimisation method.

A.1 Algorithms and Source Code Listing

Listing A.1: Count lines of C code

```
( find ./ -name '*.c' -print0 | xargs -0 cat ) | wc -l
```

A.2 Original Test Suite

The following table describes the test cases from the original test suite.

Category General

ID	Title	Description
1	Known Vulnerabilities	Check for known vulnerabilities in vulnerability databases
2	ICMP Attacks	Check for ICMP packet amplification, destination unreachable, redirect packet, blind connection reset attacks
3	IP Attacks	Check for IP spoofing attack and IP fragment overwrite attack
4	Nessus-Scan	Perform a Nessus Scan
5	Open Ports	Test for undocumented open network ports
6	Ping of Death	Test for Ping of Death vulnerability
7	SYN Flooding	Test for SYN Flooding vulnerability
8	TCP RST	Test for vulnerability with malicious TCP RST packets

Table 15: General test cases of original test suite.

Category LAN/WAN

ID	Title	Description
1	ICMP Attacks	Check for ICMP packet amplification, destination unreachable, redirect packet, blind connection reset attacks
2	IP Attacks	Check for IP spoofing attack and IP fragment overwrite attack
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	ConfigureRoutes Input Validation	Check for behaviour of the ConfigureRoutes Interface given invalid input
8	InitLAN Input Validation	Check for unwanted behaviour of the InitLAN Interface given invalid input
9	InitWAN Input Validation	Check for unwanted behaviour of the InitWAN Interface given invalid input
10	SendDataExternal Change Gateway	Test if it is possible to change the gateway via malicious data entered at SendDataExternal interface
11	SendDataExternal Input Validation	Check for unwanted behaviour of the SendDataExternal Interface given invalid input
12	SendDataHealth Client/BusinessModule Identity	Test for the possibility to masquerade as client or business module via malicious data input to the SendDataHealth interface
13	SendDataHealth Input Validation	Check for unwanted behaviour of the SendDataHealth Interface given invalid input

Table 16: Test cases for the LAN/WAN interfaces of original test suite.

Category DHCP-Client

ID	Title	Description
1	ICMP Attacks	Check for ICMP packet amplification, destination unreachable, redirect packet, blind connection reset attacks
2	IP Attacks	Check for IP spoofing attack and IP fragment overwrite attack
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	Init Input GetInformation	Check for unwanted behaviour of the Get-Information Interface given invalid input
8	Overwrite Static Network Info	Test for possible malicious input to overwrite static network information
9	GetInformation DOS	Test if it is possible to call the GetInformation script often enough in a given time period to fill the process table or cause a DoS

Table 17: Test cases for the DHCP client of original test suite.

Category DHCP-Server

ID	Title	Description
1	ICMP Attacks	Check for ICMP packet amplification, destination unreachable, redirect packet, blind connection reset attacks
2	IP Attacks	Check for IP spoofing attack and IP fragment overwrite attack
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	Init Input Validation	Check for unwanted behaviour of the Init Interface given invalid input
8	DHCP Spoofing Attack	Check for the possibility to manipulate the default gateway or name server via a DHCP spoofing attack
9	DHCP Starvation Attack	Test for the possibility to use up the whole address space via DHCP starvation attack
10	DHCP Init Stress Test	Test for possible DoS by executing the Init script repeatedly

Table 18: Test cases for the DHCP server of original test suite.

Category Name Service

ID	Title	Description
1	ICMP Attacks	Check for ICMP packet amplification, destination unreachable, redirect packet, blind connection reset attacks
2	IP Attacks	Check for IP spoofing attack and IP fragment overwrite attack
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	DNS Amplification Attack	Test for vulnerability to DNS Amplification Attack
8	DNS Spoofing	Check for vulnerability to DNS Spoofing Attack
9	GetIpAddress Input Validation	Check for unwanted behaviour of the GetIpAddress interface given invalid input
10	GetIpAddress Stress Test	Test for possible DoS attack through the GetIpAddress interface against the name service
11	GetServiceList Input Validation	Check for unwanted behaviour of the GetServiceList interface given invalid input
12	GetServiceList Stress Test	Test for possible DoS attack through the GetServiceList interface against the name service

Table 19: Name service test cases of original test suite.

Category TLS Service Incoming

ID	Title	Description

1	ICMP Attacks	Check for ICMP packet amplification, destination unreachable, redirect packet, blind connection reset attacks
2	IP Attacks	Check for IP spoofing attack and IP fragment overwrite attack
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	TLS Fuzzing	Use Fuzzing for TLS handshake
8	TLS Version Checks	Check for usage of wrong SSL versions, invalid SSL versions and try to force usage of old SSL version
9	TLS Default Version	Check if version 1.2 is set as default version on the server side
10	Allowed Ciphers	Check if only specified ciphers are allowed
11	Forbidden Ciphers	Check that forbidden ciphers are not allowed
12	Atypical Padding	Check for correct behaviour when atypical padding is applied
13	Extensions	Check that only the right TLS extensions are allowed
14	Large number of extensions	Check that only a limited number of TLS extensions is accepted
15	Fallback SCSV	Check that protocol downgrade is not possible
16	MD5 check	Check for usage of obsolete MD5 hashing
17	DHE-RSA	Check for correct key exchange signature usage
18	DHE-RSA key exchange bad message	Check for behaviour with key exchange with bad messages
19	Early application data	Check the behaviour when application data is sent before TLS connection is established

20	Early invalid application data	Check the behaviour when invalid application data is sent before TLS connection is established
21	ECDHE	Test Elliptic-Curve Diffie Hellmann Key Exchange
22	ECDHE bad message	Test Elliptic-Curve Diffie Hellmann Key Exchange with bad data
23	Empty Extension	Test for correct behaviour when empty extension is provided
24	Modified MAC	Check for correct behaviour when handshake message's MAC is modified
25	Modified Finished Message	Check for correct behaviour when handshake finished message is modified
26	Modified Cipher Text	Check for correct behaviour when cipher text is modified
27	Modified Padding	Check for correct behaviour when padding is modified
28	Client Hello	Check for correct behaviour when client hello message is modified
29	Client Hello Record Overflow	Check for correct behaviour on client hello record overflow
30	Client Hello Too Large	Check for correct behaviour with too large client hello message
31	Client Hello Truncating	Check for correct behaviour with truncated client hello message
32	Session Renegotiation	Check for session renegotiation vulnerability
33	Session Resumption	Check for correct session resumption
34	Invalid Compression	Check for use of invalid compression algorithms
35	Zero Length Data	Check for correct behaviour with zero length application data
36	Invalid Session ID	Check for correct behaviour with invalid session IDs

37	Message Duplication	Test for handling of duplicated messages
38	Message Skipping	Test for handling of skipped messages
39	Signature Algorithm Selection	Check for correct selection of signature algorithms
40	Death Alert	Test for death alert vulnerability
41	SSLv2 hello	Check for correct handling of SSLv2 hello messages
42	Finished Message Truncating	Test for correct behaviour with truncated finished messages

Table 20: Test cases for incoming connections for the TLS service of original test suite.

Category TLS Service Outgoing

ID	Title	Description
1	ICMP Attacks	Check for ICMP packet amplification, destination unreachable, redirect packet, blind connection reset attacks
2	IP Attacks	Check for IP spoofing attack and IP fragment overwrite attack
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	TLS Handshake	Check for correct TLS handshake establishing on client side
8	TLS Version Compliance	Test for correct TLS version usage
9	TLS Unknown Version Number	Test for correct behaviour on unknown version number usage
10	SSLv2 Usage	Check for correct behaviour on usage of SSLv2 data structures
11	TLSv1	Check if TLS 1.2 is default version

12	Allowed Cipher Suites	Check if client only announces allowed cipher suites
13	Server Side Change of Cipher Suites	Test how a server side change of cipher suites would be handled
14	Atypical Padding	Test how invalid padding is handled
15	Broken Extensions	Test handling of broken extensions
16	Large Number of Extensions	Test how large number of extensions would be handled
17	Session Renegotiation	Check for session renegotiation vulnerability
18	Hash and Signing Algorithms	Test for announced hashing and signing algorithms
19	Early Payload	Test for correct behaviour when early payload is being sent
20	Invalid Content	Test for correct behaviour when invalid content is being sent
21	Modified MAC	Check for correct behaviour when handshake message's MAC is modified
22	Modified Finished Message	Check for correct behaviour when handshake finished message is modified
23	Modified Cipher Text	Check for correct behaviour when cipher text is modified
24	Modified Padding	Check for correct behaviour when padding is modified
25	Session Resumption Okay	Check for correct handling of successful session resumption
26	Session Resumption Fail	Check for correct handling of failed session resumption
27	Invalid Handshake Messages	Test for correct behaviour when invalid handshake messages are sent
28	Invalid Additional Messages	Check for correct handling of additionally sent invalid messages
29	Invalid Order of Messages	Test for the handling of messages delivered in the wrong order

30	Missing Messages	Check for correct handling of missing messages
31	Padded/Truncated Finished Message	Test for correct handling of padded or truncated finished message
32	Padded/Truncated Hello Message	Test for correct handling of padded or truncated hello message
33	Padded/Truncated KeyExchange Message	Test for correct handling of padded or truncated key exchange messages
34	SSLv2 hello message	Check for proper handling of SSLv2 hello message
35	Invalid Session ID	Test for invalid session ID
36	ECDHE Cipher Support	Check support of Elliptic-Curve Diffie-Hellmann ciphers
37	Bad ECDHE Bad Messages	Test for correct behaviour given invalid ECDHE key exchange messages

Table 21: Test cases for outgoing connections for the TLS service of original test suite.

Category Time Service

ID	Title	Description
1	ICMP Attacks	Check for ICMP packet amplification, destination unreachable, redirect packet, blind connection reset attacks
2	IP Attacks	Check for IP spoofing attack and IP fragment overwrite attack
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	GetTime Input Validation	Check for unwanted behaviour of the GetTime interface given invalid input
8	GetTime Stress Test	Check behaviour of time service component given many requests on the GetTime interface
9	Init Input Validation	Check for unwanted behaviour of the Init interface given invalid input
10	Init Stress Test	Check behaviour of time service component given many requests on the Init interface
11	Spoofing	Test for the possibility of spoofing attacks in the Init interface
12	Data Manipulation	Check for the possibility of time manipulation in NTP packets
13	SyncTime Input Validation	Check for unwanted behaviour of the SyncTime interface given invalid input
14	SyncTime Stress Test	Check behaviour of time service component given many requests on the SyncTime interface
15	NTP Amplification Attack	Check for the possibility to execute an NTP amplification attack

Table 22: Test cases for time service of original test suite.

A.3 Normalized Test Suite

The following tables show the normalized test suite, in which every test case only tests for one requirement. Exceptions are a check for known vulnerabilities, where in theory many vulnerabilities are checked for, in practice, however, several databases are scanned for known issues with used software. The same applies for a Nessus scan.

Category General

ID	Title	Description
1	Known Vulnerabilities	Check for known vulnerabilities in vulnerability databases
4	Nessus-Scan	Perform a Nessus Scan
5	Open Ports	Test for undocumented open network ports
6	Ping of Death	Test for Ping of Death vulnerability
7	SYN Flooding	Test for SYN Flooding vulnerability
8	TCP RST	Test for vulnerability with malicious TCP RST packets
9	ICMP Packet Amplification Attack	Test for ICMP Packet Amplification attack
10	ICMP Destination Unreachable Attack	Test for attack with sent ICMP Destination Unreachable Packets
11	ICMP Redirect Packet Attack	Test for attack with ICMP Redirect Packets
12	ICMP Blind Connection Reset Attack	Test for ICMP Blind Connection Reset Attack
13	IP Spoofing Attack	Test for spoofed IP packets
14	IP Fragment Overwrite	Test for fragmented IP packets to manipulate protocol fields

Table 23: General test cases of normalized test suite.

Category LAN/WAN

ID	Title	Description
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	ConfigureRoutes Input Validation	Check for behaviour of the ConfigureRoutes Interface given invalid input
8	InitLAN Input Validation	Check for unwanted behaviour of the InitLAN Interface given invalid input
9	InitWAN Input Validation	Check for unwanted behaviour of the InitWAN Interface given invalid input
10	SendDataExternal Change Gateway	Test if it is possible to change the gateway via malicious data entered at SendDataExternal interface
11	SendDataExternal Input Validation	Check for unwanted behaviour of the SendDataExternal Interface given invalid input
12	SendDataHealth Client/BusinessModule Identity	Test for the possibility to masquerade as client or business module via malicious data input to the SendDataHealth interface
13	SendDataHealth Input Validation	Check for unwanted behaviour of the SendDataHealth Interface given invalid input
14	ICMP Packet Amplification Attack	Test for ICMP Packet Amplification attack
15	ICMP Destination Unreachable Attack	Test for attack with sent ICMP Destination Unreachable Packets
16	ICMP Redirect Packet Attack	Test for attack with ICMP Redirect Packets
17	ICMP Blind Connection Reset Attack	Test for ICMP Blind Connection Reset Attack
18	IP Spoofing Attack	Test for spoofed IP packets
19	IP Fragment Overwrite	Test for fragmented IP packets to manipulate protocol fields

Table 24: Test cases for the LAN/WAN interfaces of normalized test suite.

Category DHCP-Client

ID	Title	Description
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	Init Input GetInformation	Check for unwanted behaviour of the GetInformation Interface given invalid input
8	Overwrite Static Network Info	Test for possible malicious input to overwrite static network information
9	GetInformation DOS	Test if it is possible to call the GetInformation script often enough in a given time period to fill the process table or cause a DoS
10	ICMP Packet Amplification Attack	Test for ICMP Packet Amplification attack
11	ICMP Destination Unreachable Attack	Test for attack with sent ICMP Destination Unreachable Packets
12	ICMP Redirect Packet Attack	Test for attack with ICMP Redirect Packets
13	ICMP Blind Connection Reset Attack	Test for ICMP Blind Connection Reset Attack
14	IP Spoofing Attack	Test for spoofed IP packets
15	IP Fragment Overwrite	Test for fragmented IP packets to manipulate protocol fields

Table 25: Test cases for the DHCP client of normalized test suite.

Category DHCP-Server

ID	Title	Description
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	Init Input Validation	Check for unwanted behaviour of the Init Interface given invalid input
8	DHCP Spoofing Attack	Check for the possibility to manipulate the default gateway or name server via a DHCP spoofing attack
9	DHCP Starvation Attack	Test for the possibility to use up the whole address space via DHCP starvation attack
10	DHCP Init Stress Test	Test for possible DoS by executing the Init script repeatedly
11	ICMP Packet Amplification Attack	Test for ICMP Packet Amplification attack
12	ICMP Destination Unreachable Attack	Test for attack with sent ICMP Destination Unreachable Packets
13	ICMP Redirect Packet Attack	Test for attack with ICMP Redirect Packets
14	ICMP Blind Connection Reset Attack	Test for ICMP Blind Connection Reset Attack
15	IP Spoofing Attack	Test for spoofed IP packets
16	IP Fragment Overwrite	Test for fragmented IP packets to manipulate protocol fields

Table 26: Test cases for the DHCP server of normalized test suite.

Category Name Service

ID	Title	Description
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	DNS Amplification Attack	Test for vulnerability to DNS Amplification Attack
8	DNS Spoofing	Check for vulnerability to DNS Spoofing Attack
9	ICMP Packet Amplification Attack	Test for ICMP Packet Amplification attack
10	ICMP Destination Unreachable Attack	Test for attack with sent ICMP Destination Unreachable Packets
11	ICMP Redirect Packet Attack	Test for attack with ICMP Redirect Packets
12	ICMP Blind Connection Reset Attack	Test for ICMP Blind Connection Reset Attack
13	IP Spoofing Attack	Test for spoofed IP packets
14	IP Fragment Overwrite	Test for fragmented IP packets to manipulate protocol fields
15	GetIpAddress Input Validation	Check for unwanted behaviour of the GetIpAddress interface given invalid input
16	GetIpAddress Stress Test	Test for possible DoS attack through the GetIpAddress interface against the name service
17	GetServiceList Input Validation	Check for unwanted behaviour of the GetServiceList interface given invalid input
18	GetServiceList Stress Test	Test for possible DoS attack through the GetServiceList interface against the name service

Table 27: Name service test cases of normalized test suite.

Category TLS Service Incoming

ID	Title	Description
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	TLS Fuzzing	Use Fuzzing for TLS handshake
8	TLS Version Checks	Check for usage of wrong SSL versions, invalid SSL versions and try to force usage of old SSL version
9	TLS Default Version	Check if version 1.2 is set as default version on the server side
10	Allowed Ciphers	Check if only specified ciphers are allowed
11	Forbidden Ciphers	Check that forbidden ciphers are not allowed
12	Atypical Padding	Check for correct behaviour when atypical padding is applied
13	Extensions	Check that only the right TLS extensions are allowed
14	Large number of extensions	Check that only a limited number of TLS extensions is accepted
15	Fallback SCSV	Check that protocol downgrade is not possible
16	MD5 check	Check for usage of obsolete MD5 hashing
17	DHE-RSA	Check for correct key exchange signature usage
18	DHE-RSA key exchange bad message	Check for behaviour with key exchange with bad messages
19	Early application data	Check the behaviour when application data is sent before TLS connection is established

20	Early invalid application data	Check the behaviour when invalid application data is sent before TLS connection is established
21	ECDHE	Test Elliptic-Curve Diffie Hellmann Key Exchange
22	ECDHE bad message	Test Elliptic-Curve Diffie Hellmann Key Exchange with bad data
23	Empty Extension	Test for correct behaviour when empty extension is provided
24	Modified MAC	Check for correct behaviour when handshake message's MAC is modified
25	Modified Finished Message	Check for correct behaviour when handshake finished message is modified
26	Modified Cipher Text	Check for correct behaviour when cipher text is modified
27	Modified Padding	Check for correct behaviour when padding is modified
28	Client Hello	Check for correct behaviour when client hello message is modified
29	Client Hello Record Overflow	Check for correct behaviour on client hello record overflow
30	Client Hello Too Large	Check for correct behaviour with too large client hello message
31	Client Hello Truncating	Check for correct behaviour with truncated client hello message
32	Session Renegotiation	Check for session renegotiation vulnerability
33	Session Resumption	Check for correct session resumption
34	Invalid Compression	Check for use of invalid compression algorithms
35	Zero Length Data	Check for correct behaviour with zero length application data
36	Invalid Session ID	Check for correct behaviour with invalid session IDs

37	Message Duplication	Test for handling of duplicated messages
38	Message Skipping	Test for handling of skipped messages
39	Signature Algorithm Selection	Check for correct selection of signature algorithms
40	Death Alert	Test for death alert vulnerability
41	SSLv2 hello	Check for correct handling of SSLv2 hello messages
42	Finished Message Truncating	Test for correct behaviour with truncated finished messages
43	ICMP Packet Amplification Attack	Test for ICMP Packet Amplification attack
44	ICMP Destination Unreachable Attack	Test for attack with sent ICMP Destination Unreachable Packets
45	ICMP Redirect Packet Attack	Test for attack with ICMP Redirect Packets
46	ICMP Blind Connection Reset Attack	Test for ICMP Blind Connection Reset Attack
47	IP Spoofing Attack	Test for spoofed IP packets
48	IP Fragment Overwrite	Test for fragmented IP packets to manipulate protocol fields

Table 28: Test cases for incoming connections for the TLS service of normalized test suite.

Category TLS Service Outgoing

ID	Title	Description
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	TLS Handshake	Check for correct TLS handshake establishing on client side

8	TLS Version Compliance	Test for correct TLS version usage
9	TLS Unknown Version Number	Test for correct behaviour on unknown version number usage
10	SSLv2 Usage	Check for correct behaviour on usage of SSLv2 data structures
11	TLSv1	Check if TLS 1.2 is default version
12	Allowed Cipher Suites	Check if client only announces allowed cipher suites
13	Server Side Change of Cipher Suites	Test how a server side change of cipher suites would be handled
14	Atypical Padding	Test how invalid padding is handled
15	Broken Extensions	Test handling of broken extensions
16	Large Number of Extensions	Test how large number of extensions would be handled
17	Session Renegotiation	Check for session renegotiation vulnerability
18	Hash and Signing Algorithms	Test for announced hashing and signing algorithms
19	Early Payload	Test for correct behaviour when early payload is being sent
20	Invalid Content	Test for correct behaviour when invalid content is being sent
21	Modified MAC	Check for correct behaviour when handshake message's MAC is modified
22	Modified Finished Message	Check for correct behaviour when handshake finished message is modified
23	Modified Cipher Text	Check for correct behaviour when cipher text is modified
24	Modified Padding	Check for correct behaviour when padding is modified
25	Session Resumption Okay	Check for correct handling of successful session resumption
26	Session Resumption Fail	Check for correct handling of failed session resumption

27	Invalid Handshake Messages	Test for correct behaviour when invalid handshake messages are sent
28	Invalid Additional Messages	Check for correct handling of additionally sent invalid messages
29	Invalid Order of Messages	Test for the handling of messages delivered in the wrong order
30	Missing Messages	Check for correct handling of missing messages
31	Padded/Truncated Finished Message	Test for correct handling of padded or truncated finished message
32	Padded/Truncated Hello Message	Test for correct handling of padded or truncated hello message
33	Padded/Truncated KeyExchange Message	Test for correct handling of padded or truncated key exchange messages
34	SSLv2 hello message	Check for proper handling of SSLv2 hello message
35	Invalid Session ID	Test for invalid session ID
36	ECDHE Cipher Support	Check support of Elliptic-Curve Diffie-Hellmann ciphers
37	Bad ECDHE Bad Messages	Test for correct behaviour given invalid ECDHE key exchange messages
38	ICMP Packet Amplification Attack	Test for ICMP Packet Amplification attack
39	ICMP Destination Unreachable Attack	Test for attack with sent ICMP Destination Unreachable Packets
40	ICMP Redirect Packet Attack	Test for attack with ICMP Redirect Packets
41	ICMP Blind Connection Reset Attack	Test for ICMP Blind Connection Reset Attack
42	IP Spoofing Attack	Test for spoofed IP packets
43	IP Fragment Overwrite	Test for fragmented IP packets to manipulate protocol fields

Table 29: Test cases for outgoing connections for the TLS service of normalized test suite.

Category Time Service

ID	Title	Description
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	GetTime Input Validation	Check for unwanted behaviour of the GetTime interface given invalid input
8	GetTime Stress Test	Check behaviour of time service component given many requests on the GetTime interface
9	Init Input Validation	Check for unwanted behaviour of the Init interface given invalid input
10	Init Stress Test	Check behaviour of time service component given many requests on the Init interface
11	Spoofing	Test for the possibility of spoofing attacks in the Init interface
12	Data Manipulation	Check for the possibility of time manipulation in NTP packets
13	SyncTime Input Validation	Check for unwanted behaviour of the SyncTime interface given invalid input
14	SyncTime Stress Test	Check behaviour of time service component given many requests on the SyncTime interface
15	NTP Amplification Attack	Check for the possibility to execute an NTP amplification attack
16	ICMP Packet Amplification Attack	Test for ICMP Packet Amplification attack
17	ICMP Destination Unreachable Attack	Test for attack with sent ICMP Destination Unreachable Packets
18	ICMP Redirect Packet Attack	Test for attack with ICMP Redirect Packets
19	ICMP Blind Connection Reset Attack	Test for ICMP Blind Connection Reset Attack
20	IP Spoofing Attack	Test for spoofed IP packets
21	IP Fragment Overwrite	Test for fragmented IP packets to manipulate protocol fields

Table 30: Test cases for time service of normalized test suite.

A.4 Reduced Test Suite

This section shows the test suite after the test suite reduction has been applied. In comparison to the normalised test suite from Appendix , many network specific tests could be removed, as the SUT's subcomponents share the same network stack.

Category General

ID	Title	Description
1	Known Vulnerabilities	Check for known vulnerabilities in vulnerability databases
4	Nessus-Scan	Perform a Nessus Scan

Table 31: General test cases of reduced test suite.

Category LAN/WAN

ID	Title	Description
3	Open Ports	Test for undocumented open network ports
4	Ping of Death	Test for Ping of Death vulnerability
5	SYN Flooding	Test for SYN Flooding vulnerability
6	TCP RST	Test for vulnerability with malicious TCP RST packets
7	ConfigureRoutes Input Validation	Check for behaviour of the ConfigureRoutes Interface given invalid input
8	InitLAN Input Validation	Check for unwanted behaviour of the InitLAN Interface given invalid input
9	InitWAN Input Validation	Check for unwanted behaviour of the InitWAN Interface given invalid input
10	SendDataExternal Change Gateway	Test if it is possible to change the gateway via malicious data entered at SendDataExternal interface
11	SendDataExternal Input Validation	Check for unwanted behaviour of the SendDataExternal Interface given invalid input
12	SendDataHealth Client/BusinessModule Identity	Test for the possibility to masquerade as client or business module via malicious data input to the SendDataHealth interface
13	SendDataHealth Input Validation	Check for unwanted behaviour of the SendDataHealth Interface given invalid input
14	ICMP Packet Amplification Attack	Test for ICMP Packet Amplification attack
15	ICMP Destination Unreachable Attack	Test for attack with sent ICMP Destination Unreachable Packets
16	ICMP Redirect Packet Attack	Test for attack with ICMP Redirect Packets
17	ICMP Blind Connection Reset Attack	Test for ICMP Blind Connection Reset Attack
18	IP Spoofing Attack	Test for spoofed IP packets
19	IP Fragment Overwrite	Test for fragmented IP packets to manipulate protocol fields

Table 32: Test cases for the LAN/WAN interfaces of reduced test suite.

Category DHCP-Client

ID	Title	Description
7	Init Input GetInformation	Check for unwanted behaviour of the GetInformation Interface given invalid input
8	Overwrite Static Network Info	Test for possible malicious input to overwrite static network information
9	GetInformation DOS	Test if it is possible to call the GetInformation script often enough in a given time period to fill the process table or cause a DoS

Table 33: Test cases for the DHCP client of reduced test suite.

Category DHCP-Server

ID	Title	Description
7	Init Input Validation	Check for unwanted behaviour of the Init Interface given invalid input
8	DHCP Spoofing Attack	Check for the possibility to manipulate the default gateway or name server via a DHCP spoofing attack
9	DHCP Starvation Attack	Test for the possibility to use up the whole address space via DHCP starvation attack
10	DHCP Init Stress Test	Test for possible DoS by executing the Init script repeatedly

Table 34: Test cases for the DHCP server of reduced test suite.

Category Name Service

ID	Title	Description
7	DNS Amplification Attack	Test for vulnerability to DNS Amplification Attack
8	DNS Spoofing	Check for vulnerability to DNS Spoofing Attack
15	GetIpAddress Input Validation	Check for unwanted behaviour of the GetIpAddress interface given invalid input
16	GetIpAddress Stress Test	Test for possible DoS attack through the GetIpAddress interface against the name service
17	GetServiceList Input Validation	Check for unwanted behaviour of the GetServiceList interface given invalid input
18	GetServiceList Stress Test	Test for possible DoS attack through the GetServiceList interface against the name service

Table 35: Name service test cases of reduced test suite.

Category TLS Service Incoming

ID	Title	Description
7	TLS Fuzzing	Use Fuzzing for TLS handshake
8	TLS Version Checks	Check for usage of wrong SSL versions, invalid SSL versions and try to force usage of old SSL version
9	TLS Default Version	Check if version 1.2 is set as default version on the server side
10	Allowed Ciphers	Check if only specified ciphers are allowed
11	Forbidden Ciphers	Check that forbidden ciphers are not allowed
12	Atypical Padding	Check for correct behaviour when atypical padding is applied

13	Extensions	Check that only the right TLS extensions are allowed
14	Large number of extensions	Check that only a limited number of TLS extensions is accepted
15	Fallback SCSV	Check that protocol downgrade is not possible
16	MD5 check	Check for usage of obsolete MD5 hashing
17	DHE-RSA	Check for correct key exchange signature usage
18	DHE-RSA key exchange bad message	Check for behaviour with key exchange with bad messages
19	Early application data	Check the behaviour when application data is sent before TLS connection is established
20	Early invalid application data	Check the behaviour when invalid application data is sent before TLS connection is established
21	ECDHE	Test Elliptic-Curve Diffie Hellmann Key Exchange
22	ECDHE bad message	Test Elliptic-Curve Diffie Hellmann Key Exchange with bad data
23	Empty Extension	Test for correct behaviour when empty extension is provided
24	Modified MAC	Check for correct behaviour when handshake message's MAC is modified
25	Modified Finished Message	Check for correct behaviour when handshake finished message is modified
26	Modified Cipher Text	Check for correct behaviour when cipher text is modified
27	Modified Padding	Check for correct behaviour when padding is modified
28	Client Hello	Check for correct behaviour when client hello message is modified
29	Client Hello Record Overflow	Check for correct behaviour on client hello record overflow

30	Client Hello Too Large	Check for correct behaviour with too large client hello message
31	Client Hello Truncating	Check for correct behaviour with truncated client hello message
32	Session Renegotiation	Check for session renegotiation vulnerability
33	Session Resumption	Check for correct session resumption
34	Invalid Compression	Check for use of invalid compression algorithms
35	Zero Length Data	Check for correct behaviour with zero length application data
36	Invalid Session ID	Check for correct behaviour with invalid session IDs
37	Message Duplication	Test for handling of duplicated messages
38	Message Skipping	Test for handling of skipped messages
39	Signature Algorithm Selection	Check for correct selection of signature algorithms
40	Death Alert	Test for death alert vulnerability
41	SSLv2 hello	Check for correct handling of SSLv2 hello messages
42	Finished Message Truncating	Test for correct behaviour with truncated finished messages

Table 36: Test cases for incoming connections for the TLS service of reduced test suite.

Category TLS Service Outgoing

ID	Title	Description
7	TLS Handshake	Check for correct TLS handshake establishing on client side
8	TLS Version Compliance	Test for correct TLS version usage
9	TLS Unknown Version Number	Test for correct behaviour on unknown version number usage

10	SSLv2 Usage	Check for correct behaviour on usage of SSLv2 data structures
11	TLSv1	Check if TLS 1.2 is default version
12	Allowed Cipher Suites	Check if client only announces allowed cipher suites
13	Server Side Change of Cipher Suites	Test how a server side change of cipher suites would be handled
14	Atypical Padding	Test how invalid padding is handled
15	Broken Extensions	Test handling of broken extensions
16	Large Number of Extensions	Test how large number of extensions would be handled
17	Session Renegotiation	Check for session renegotiation vulnerability
18	Hash and Signing Algorithms	Test for announced hashing and signing algorithms
19	Early Payload	Test for correct behaviour when early payload is being sent
20	Invalid Content	Test for correct behaviour when invalid content is being sent
21	Modified MAC	Check for correct behaviour when handshake message's MAC is modified
22	Modified Finished Message	Check for correct behaviour when handshake finished message is modified
23	Modified Cipher Text	Check for correct behaviour when cipher text is modified
24	Modified Padding	Check for correct behaviour when padding is modified
25	Session Resumption Okay	Check for correct handling of successful session resumption
26	Session Resumption Fail	Check for correct handling of failed session resumption
27	Invalid Handshake Messages	Test for correct behaviour when invalid handshake messages are sent

28	Invalid Additional Messages	Check for correct handling of additionally sent invalid messages
29	Invalid Order of Messages	Test for the handling of messages delivered in the wrong order
30	Missing Messages	Check for correct handling of missing messages
31	Padded/Truncated Finished Message	Test for correct handling of padded or truncated finished message
32	Padded/Truncated Hello Message	Test for correct handling of padded or truncated hello message
33	Padded/Truncated KeyExchange Message	Test for correct handling of padded or truncated key exchange messages
34	SSLv2 hello message	Check for proper handling of SSLv2 hello message
35	Invalid Session ID	Test for invalid session ID
36	ECDHE Cipher Support	Check support of Elliptic-Curve Diffie-Hellmann ciphers
37	Bad ECDHE Bad Messages	Test for correct behaviour given invalid ECDHE key exchange messages

Table 37: Test cases for outgoing connections for the TLS service of reduced test suite.

Category Time Service

ID	Title	Description
7	GetTime Input Validation	Check for unwanted behaviour of the GetTime interface given invalid input
8	GetTime Stress Test	Check behaviour of time service component given many requests on the GetTime interface
9	Init Input Validation	Check for unwanted behaviour of the Init interface given invalid input
10	Init Stress Test	Check behaviour of time service component given many requests on the Init interface
11	Spoofing	Test for the possibility of spoofing attacks in the Init interface
12	Data Manipulation	Check for the possibility of time manipulation in NTP packets
13	SyncTime Input Validation	Check for unwanted behaviour of the SyncTime interface given invalid input
14	SyncTime Stress Test	Check behaviour of time service component given many requests on the SyncTime interface
15	NTP Amplification Attack	Check for the possibility to execute an NTP amplification attack

Table 38: Test cases for time service of reduced test suite.

A.5 Partially Automated Test Suite

The tables in this section depict the reduced test suite including cost information and the decision whether to automate the test cases or not. This is calculated according to Section 3.4, with the number of executions of each test case being 2.

Category General

ID	Title	Auto. Cost	Auto. Exec. Cost	Manual Exec. Cost	Autom.
1	Known Vulnerabilities	8	1	2	NO
4	Nessus-Scan	0	1	50	YES

Table 39: General test cases with automation information.

Category LAN/WAN

ID	Title	Auto. Cost	Auto. Exec. Cost	Manual Exec. Cost	Autom.
3	Open Ports	2	0,25	1	NO
4	Ping of Death	2	0,25	1	NO
5	SYN Flooding	2	0,25	1	NO
6	TCP RST	1,5	0,25	1	YES
7	ConfigureRoutes Input Validation	5	0,25	2	NO
8	InitLAN Input Validation	5	0,25	2	NO
9	InitWAN Input Validation	5	0,25	2	NO
10	SendDataExternal Change Gateway	8	1	5	YES
11	SendDataExternal Input Validation	5	0,25	2	NO
12	SendDataHealth Client/BusinessModule Identity	16	1	9	YES
13	SendDataHealth Input Validation	5	0,25	2	NO
14	ICMP Packet Amplification Attack	2	0,25	1	NO
15	ICMP Destination Unreachable Attack	2	0,25	1	NO
16	ICMP Redirect Packet Attack	2	0,25	1	NO
17	ICMP Blind Connection Reset Attack	2	0,25	1	NO
18	IP Spoofing Attack	8	0,5	6	YES
19	IP Fragment Overwrite	16	1	8	NO

Table 40: LAN/WAN test cases with automation information.

Category DHCP-Client

ID	Title	Auto. Cost	Auto. Exec. Cost	Manual Exec. Cost	Autom.
7	Init Input GetInformation	5	0,25	2	NO
8	Overwrite Static Network Info	14	1	12	YES
9	GetInformation DOS	3	0,25	2	YES

Table 41: DHCP client test cases with automation information.

Category DHCP-Server

ID	Title	Auto. Cost	Auto. Exec. Cost	Manual Exec. Cost	Autom.
7	Init Input Validation	5	0,25	2	NO
8	DHCP Spoofing Attack	8	0,5	6	YES
9	DHCP Starvation Attack	3	0,25	2,5	YES
10	DHCP Init Stress Test	1	0,25	0,75	YES

Table 42: DHCP server test cases with automation information.

Category Name Service

ID	Title	Auto. Cost	Auto. Exec. Cost	Manual Exec. Cost	Autom.
7	DNS Amplification Attack	3	0,5	2	YES
8	DNS Spoofing	8	0,5	6	YES
15	GetIpAddress Input Validation	5	0,25	2	NO
16	GetIpAddress Stress Test	1	0,25	0,75	YES
17	GetServiceList Input Validation	5	0,25	2	NO
18	GetServiceList Stress Test	1	0,25	0,75	YES

Table 43: Name service test cases with automation information.

Category TLS Service Incoming

ID	Title	Auto. Cost	Auto. Exec. Cost	Manual Exec. Cost	Autom.
7	TLS Fuzzing	2	0,5	1,5	YES
8	TLS Version Checks	1	0,25	0,5	NO
9	TLS Default Version	1	0,25	0,5	NO
10	Allowed Ciphers	2	0,25	1	NO
11	Forbidden Ciphers	2	0,25	1	NO
12	Atypical Padding	3	0,25	2	YES
13	Extensions	2	0,25	1	NO
14	Large number of extensions	2	0,25	1	NO
15	Fallback SCSV	4	0,25	3	YES
16	MD5 check	1,5	0,25	1	YES
17	DHE-RSA	3	0,25	2	YES

18	DHE-RSA key exchange bad message	3	0,25	2	YES
19	Early application data	2	0,25	1	NO
20	Early invalid application data	2,5	0,25	1,5	YES
21	ECDHE	4	0,5	2	NO
22	ECDHE bad message	4	0,5	2	NO
23	Empty Extension	3	0,25	1,5	NO
24	Modified MAC	2	0,25	1,5	YES
25	Modified Finished Message	2	0,25	1,5	YES
26	Modified Cipher Text	2	0,25	1,5	YES
27	Modified Padding	2	0,25	1,5	YES
28	Client Hello	2	0,25	1,5	YES
29	Client Hello Record Overflow	2	0,25	1	NO
30	Client Hello Too Large	2	0,25	1	NO
31	Client Hello Truncating	2	0,25	1	NO
32	Session Renegotiation	4	0,5	2	NO
33	Session Resumption	4	0,5	2	NO
34	Invalid Compression	1,5	0,25	1	YES
35	Zero Length Data	1,5	0,25	1	YES
36	Invalid Session ID	2	0,25	1,5	YES
37	Message Duplication	3	0,25	1,5	NO
38	Message Skipping	3	0,25	1,5	NO
39	Signature Algorithm Selection	2	0,25	1	NO
40	Death Alert	1,5	0,25	1	YES
41	SSLv2 hello	2	0,25	1	NO
42	Finished Message Truncating	5	0,25	2	NO

Table 44: TLS service incoming test cases with automation information.

Category TLS Service Outgoing

ID	Title	Auto. Cost	Auto. Exec. Cost	Manual Exec. Cost	Autom.
7	TLS Handshake	2	0,5	1,5	YES
8	TLS Version Compliance	1	0,25	0,5	NO
9	TLS Unknown Version Number	1,5	0,25	0,5	NO
10	SSLv2 Usage	2	0,25	1	NO
11	TLSv1	1	0,25	0,5	NO
12	Allowed Cipher Suites	1	0,25	0,5	NO
13	Server Side Change of Cipher Suites	3	0,5	1,5	NO
14	Atypical Padding	3	0,25	1	NO
15	Broken Extensions	2	0,25	1	NO
16	Large Number of Extensions	2	0,25	1	NO
17	Session Renegotiation	5	0,25	3	YES
18	Hash and Signing Algorithms	2	0,25	1	NO
19	Early Payload	2	0,25	1	NO
20	Invalid Content	2,5	0,25	1,5	YES
21	Modified MAC	2	0,25	1,5	YES
22	Modified Finished Message	2	0,25	1,5	YES
23	Modified Cipher Text	2	0,25	1,5	YES
24	Modified Padding	2	0,25	1,5	YES
25	Session Resumption Okay	4	0,5	2	NO

26	Session Resumption Fail	4	0,5	2	NO
27	Invalid Handshake Messages	2	0,5	1,5	YES
28	Invalid Additional Messages	2	0,5	1,5	YES
29	Invalid Order of Messages	3	0,5	2	YES
30	Missing Messages	3	0,5	2	YES
31	Padded/Truncated Finished Message	2	0,5	1	NO
32	Padded/Truncated Hello Message	2	0,5	1	NO
33	Padded/Truncated KeyExchange Message	2	0,5	1	NO
34	SSLv2 hello message	2	0,25	1	NO
35	Invalid Session ID	2	0,25	1,5	YES
36	ECDHE Cipher Support	2	0,25	1	NO
37	Bad ECDHE Bad Messages	3	0,25	2	YES

Table 45: TLS service outgoing test cases with automation information.

Category Time Service

ID	Title	Auto. Cost	Auto. Exec. Cost	Manual Exec. Cost	Autom.
7	GetTime Input Validation	5	0,25	2	NO
8	GetTime Stress Test	2	0,25	1	NO
9	Init Input Validation	5	0,25	2	NO
10	Init Stress Test	2	0,25	1	NO
11	Spoofing	4	0,25	2	NO
12	Data Manipulation	3	0,25	2	YES
13	SyncTime Input Validation	5	0,25	2	NO
14	SyncTime Stress Test	2	0,25	1	NO
15	NTP Amplification Attack	4	0,25	2	NO

Table 46: Time service test cases with automation information.

A.6 Prioritised Test Suite

This section depicts the reduced and partially automated test suite ranked by calculated priority score according to Equation 3.25. The value for ‘Cost in EUR’ has been determined by the cost in hours multiplied by 100 EUR/hour. The cost in hour is, if the test case has been chosen for automation,

$$CostOfAutomation + CostOfAutomaticExecution * NumberOfExecutions$$

or otherwise

$$CostOfManualExecution * NumberOfExecutions$$

ID	Title	Category	Priority Score	Cost in EUR
1	Known Vulnerabilities	General	6,0625	400
4	Nessus-Scan	General	6,0625	200
18	IP Spoofing Attack	LAN/WAN	5,5	900
7	TLS Fuzzing	TLS Service Incoming	5,4375	300

8	TLS Version Checks	TLS Service Incoming	5,25	100
4	Ping of Death	LAN/WAN	5,125	200
8	DNS Spoofing	Name Service	5,125	900
12	Allowed Cipher Suites	TLS Service Outgoing	5,125	100
21	Modified MAC	TLS Service Outgoing	5,0625	250
8	Overwrite Static Network Info	DHCP-Client	5	1600
8	DHCP Spoofing Attack	DHCP-Server	5	900
12	Atypical Padding	TLS Service Incoming	5	350
16	MD5 check	TLS Service Incoming	5	200
10	SSLv2 Usage	TLS Service Outgoing	5	200
11	TLSv1	TLS Service Outgoing	5	100
39	Signature Algorithm Selection	TLS Service Incoming	4,9375	200
13	Server Side Change of Cipher Suites	TLS Service Outgoing	4,9375	300
15	Broken Extensions	TLS Service Outgoing	4,9375	200
10	Allowed Ciphers	TLS Service Incoming	4,875	200
11	Forbidden Ciphers	TLS Service Incoming	4,875	200
24	Modified MAC	TLS Service Incoming	4,875	250
8	TLS Version Compliance	TLS Service Outgoing	4,875	100
9	TLS Unknown Version Number	TLS Service Outgoing	4,875	100
23	Modified Cipher Text	TLS Service Outgoing	4,875	250
3	Open Ports	LAN/WAN	4,8125	200
20	Early invalid application data	TLS Service Incoming	4,8125	300
28	Client Hello	TLS Service Incoming	4,8125	250
9	TLS Default Version	TLS Service Incoming	4,75	100
21	ECDHE	TLS Service Incoming	4,75	400

25	Modified Finished Message	TLS Service Incoming	4,75	250
26	Modified Cipher Text	TLS Service Incoming	4,75	250
27	Modified Padding	TLS Service Incoming	4,75	250
31	Client Hello Truncating	TLS Service Incoming	4,75	200
37	Message Duplication	TLS Service Incoming	4,75	300
38	Message Skipping	TLS Service Incoming	4,75	300
42	Finished Message Truncating	TLS Service Incoming	4,75	400
14	Atypical Padding	TLS Service Outgoing	4,75	200
18	Hash and Signing Algorithms	TLS Service Outgoing	4,75	200
22	Modified Finished Message	TLS Service Outgoing	4,75	250
24	Modified Padding	TLS Service Outgoing	4,75	250
27	Invalid Handshake Messages	TLS Service Outgoing	4,75	300
28	Invalid Additional Messages	TLS Service Outgoing	4,75	300
29	Invalid Order of Messages	TLS Service Outgoing	4,75	400
31	Padded/Truncated Finished Message	TLS Service Outgoing	4,75	200
32	Padded/Truncated Hello Message	TLS Service Outgoing	4,75	200
33	Padded/Truncated KeyExchange Message	TLS Service Outgoing	4,75	200
36	ECDHE Cipher Support	TLS Service Outgoing	4,75	200
37	Bad ECDHE Bad Messages	TLS Service Outgoing	4,75	350
19	Early application data	TLS Service Incoming	4,6875	200
32	Session Renegotiation	TLS Service Incoming	4,6875	400

34	Invalid Compression	TLS Service Incoming	4,6875	200
35	Zero Length Data	TLS Service Incoming	4,6875	200
41	SSLv2 hello	TLS Service Incoming	4,6875	200
19	Early Payload	TLS Service Outgoing	4,6875	200
20	Invalid Content	TLS Service Outgoing	4,6875	300
5	SYN Flooding	LAN/WAN	4,625	200
6	TCP RST	LAN/WAN	4,625	200
13	Extensions	TLS Service Incoming	4,625	200
29	Client Hello Record Overflow	TLS Service Incoming	4,625	200
33	Session Resumption	TLS Service Incoming	4,625	400
36	Invalid Session ID	TLS Service Incoming	4,625	250
35	Invalid Session ID	TLS Service Outgoing	4,625	250
12	Data Manipulation	Time Service	4,625	350
19	IP Fragment Overwrite	LAN/WAN	4,5625	1600
9	DHCP Starvation Attack	DHCP-Server	4,5625	350
7	DNS Amplification Attack	Name Service	4,5625	400
23	Empty Extension	TLS Service Incoming	4,5	300
7	TLS Handshake	TLS Service Outgoing	4,5	300
16	Large Number of Extensions	TLS Service Outgoing	4,5	200
25	Session Resumption Okay	TLS Service Outgoing	4,5	400
26	Session Resumption Fail	TLS Service Outgoing	4,5	400
17	DHE-RSA	TLS Service Incoming	4,4375	350
18	DHE-RSA key exchange bad message	TLS Service Incoming	4,4375	350
22	ECDHE bad message	TLS Service Incoming	4,4375	400
30	Missing Messages	TLS Service Outgoing	4,4375	400

34	SSLv2 hello message	TLS Service Outgoing	4,4375	200
7	ConfigureRoutes Input Validation	LAN/WAN	4,375	400
15	Fallback SCSV	TLS Service Incoming	4,375	450
11	Spoofing	Time Service	4,375	400
11	SendDataExternal Input Validation	LAN/WAN	4,3125	400
30	Client Hello Too Large	TLS Service Incoming	4,3125	200
10	SendDataExternal Change Gateway	LAN/WAN	4,1875	1000
14	ICMP Packet Amplification Attack	LAN/WAN	4,1875	200
40	Death Alert	TLS Service Incoming	4,125	200
17	Session Renegotiation	TLS Service Outgoing	4,125	550
15	ICMP Destination Unreachable Attack	LAN/WAN	4,0625	200
16	ICMP Redirect Packet Attack	LAN/WAN	4,0625	200
17	ICMP Blind Connection Reset Attack	LAN/WAN	4,0625	200
15	NTP Amplification Attack	Time Service	3,875	400
13	SyncTime Input Validation	Time Service	3,8125	400
8	InitLAN Input Validation	LAN/WAN	3,75	400
9	InitWAN Input Validation	LAN/WAN	3,75	400
12	SendDataHealth Client/BusinessModule Identity	LAN/WAN	3,6875	1800
13	SendDataHealth Input Validation	LAN/WAN	3,625	400

10	DHCP Init Stress Test	DHCP-Server	3,625	150
7	GetTime Input Validation	Time Service	3,625	400
9	Init Input Validation	Time Service	3,625	400
14	Large number of extensions	TLS Service Incoming	3,5625	200
7	Init Input Validation	DHCP-Server	3,375	400
7	Init Input GetInformation	DHCP-Client	3,1875	400
9	GetInformation DOS	DHCP-Client	3,125	350
15	GetIpAddress Input Validation	Name Service	3,125	400
17	GetServiceList Input Validation	Name Service	3,125	400
16	GetIpAddress Stress Test	Name Service	3	150
16	GetServiceList Stress Test	Name Service	3	150
8	GetTime Stress Test	Time Service	2,75	200
10	Init Stress Test	Time Service	2,75	200
14	SyncTime Stress Test	Time Service	2,625	200

Table 47: Prioritised test suite.

A.7 Final Test Suite

This section shows the final test suite after application of the test case prioritisation step of the optimisation method and applying the budgetary constraint of EUR 20.000.

ID	Title	Category	Priority Score	Cost in EUR
1	Known Vulnerabilities	General	6,0625	400
4	Nessus-Scan	General	6,0625	200
18	IP Spoofing Attack	LAN/WAN	5,5	900

7	TLS Fuzzing	TLS Service Incoming	5,4375	300
8	TLS Version Checks	TLS Service Incoming	5,25	100
4	Ping of Death	LAN/WAN	5,125	200
8	DNS Spoofing	Name Service	5,125	900
12	Allowed Cipher Suites	TLS Service Outgoing	5,125	100
21	Modified MAC	TLS Service Outgoing	5,0625	250
8	Overwrite Static Network Info	DHCP-Client	5	1600
8	DHCP Spoofing Attack	DHCP-Server	5	900
12	Atypical Padding	TLS Service Incoming	5	350
16	MD5 check	TLS Service Incoming	5	200
10	SSLv2 Usage	TLS Service Outgoing	5	200
11	TLSv1	TLS Service Outgoing	5	100
39	Signature Algorithm Selection	TLS Service Incoming	4,9375	200
13	Server Side Change of Cipher Suites	TLS Service Outgoing	4,9375	300
15	Broken Extensions	TLS Service Outgoing	4,9375	200
10	Allowed Ciphers	TLS Service Incoming	4,875	200
11	Forbidden Ciphers	TLS Service Incoming	4,875	200
24	Modified MAC	TLS Service Incoming	4,875	250
8	TLS Version Compliance	TLS Service Outgoing	4,875	100
9	TLS Unknown Version Number	TLS Service Outgoing	4,875	100
23	Modified Cipher Text	TLS Service Outgoing	4,875	250
3	Open Ports	LAN/WAN	4,8125	200
20	Early invalid application data	TLS Service Incoming	4,8125	300
28	Client Hello	TLS Service Incoming	4,8125	250
9	TLS Default Version	TLS Service Incoming	4,75	100

21	ECDHE	TLS Service Incoming	4,75	400
25	Modified Finished Message	TLS Service Incoming	4,75	250
26	Modified Cipher Text	TLS Service Incoming	4,75	250
27	Modified Padding	TLS Service Incoming	4,75	250
31	Client Hello Truncating	TLS Service Incoming	4,75	200
37	Message Duplication	TLS Service Incoming	4,75	300
38	Message Skipping	TLS Service Incoming	4,75	300
42	Finished Message Truncating	TLS Service Incoming	4,75	400
14	Atypical Padding	TLS Service Outgoing	4,75	200
18	Hash and Signing Algorithms	TLS Service Outgoing	4,75	200
22	Modified Finished Message	TLS Service Outgoing	4,75	250
24	Modified Padding	TLS Service Outgoing	4,75	250
27	Invalid Handshake Messages	TLS Service Outgoing	4,75	300
28	Invalid Additional Messages	TLS Service Outgoing	4,75	300
29	Invalid Order of Messages	TLS Service Outgoing	4,75	400
31	Padded/Truncated Finished Message	TLS Service Outgoing	4,75	200
32	Padded/Truncated Hello Message	TLS Service Outgoing	4,75	200
33	Padded/Truncated KeyExchange Message	TLS Service Outgoing	4,75	200
36	ECDHE Cipher Support	TLS Service Outgoing	4,75	200
37	Bad ECDHE Bad Messages	TLS Service Outgoing	4,75	350
19	Early application data	TLS Service Incoming	4,6875	200

32	Session Renegotiation	TLS Service Incoming	4,6875	400
34	Invalid Compression	TLS Service Incoming	4,6875	200
35	Zero Length Data	TLS Service Incoming	4,6875	200
41	SSLv2 hello	TLS Service Incoming	4,6875	200
19	Early Payload	TLS Service Outgoing	4,6875	200
20	Invalid Content	TLS Service Outgoing	4,6875	300
5	SYN Flooding	LAN/WAN	4,625	200
6	TCP RST	LAN/WAN	4,625	200
13	Extensions	TLS Service Incoming	4,625	200
29	Client Hello Record Overflow	TLS Service Incoming	4,625	200
33	Session Resumption	TLS Service Incoming	4,625	400
36	Invalid Session ID	TLS Service Incoming	4,625	250
35	Invalid Session ID	TLS Service Outgoing	4,625	250
12	Data Manipulation	Time Service	4,625	350
9	DHCP Starvation Attack	DHCP-Server	4,5625	350
7	DNS Amplification Attack	Name Service	4,5625	400
23	Empty Extension	TLS Service Incoming	4,5	300
7	TLS Handshake	TLS Service Outgoing	4,5	300
10	DHCP Init Stress Test	DHCP-Server	3,625	150

Table 48: The final test suite considering a given budget of EUR 20.000.

List of Figures

List of Tables

Acronyms

API	Application Programming Interface. 37, 38
AST	Automated Software Testing. 67
BSI	Bundesamt für Sicherheit in der Informationstechnik. 14, 40, 50, 53, 54, 149
CIA	Confidentiality, Integrity, Availability. 6
CNSS	Committee on National Security Systems. 6
DAC	Discretionary Access Control. 8
DDoS	Distributed Denial of Service. 10
DNS	Domain Name System. 84
DoS	Denial of Service. 10, 36, 104–106, 115–117, 126, 127
eGK	elektronische Gesundheitskarte. 51
eHBA	elektronischer Heilberufsausweis. 51
ELGA	Elektronische Gesundheitsakte. 46
eMP	elektronische Medikationsplan. 51
ePA	elektronische Patientenakte. 51
EUGDPR	EU General Data Protection Regulation. 10, 12, 14, 53
IEEE	Institute of Electrical and Electronics Engineers. 26
ISMS	Information Security Management System. 48
ISO	International Organization for Standardization. 5, 20, 48
ISP	Internet Service Provider. 10

KBV Kassenärztliche Bundesvereinigung. 50

MAC Mandatory Access Control. 7, 8

MITM Man in the Middle. 9

NFDM Notfalldatenmanagement. 51

NIST National Institute of Standards and Technology. 5, 8, 42–44, 56, 149

OWASP Open Web Application Security Project. 77, 82, 99

PTES Penetration Testing Execution Standard. 56

ROI Return of Investment. 69, 70

SDL Security Development Lifecycle. 23, 24

SMC-B Security Module Card Type B. 51

STI Sexually Transmitted Infection. 47

SUT System Under Test. 1–3, 24, 29, 30, 36–39, 41, 43, 44, 55–57, 59, 60, 62, 68, 69, 74, 75, 83, 86, 87, 100, 124

TADM Test Automation Decision Matrix. 70

TI Telematik Infrastructure. 50, 51

TOE Target of Evaluation. 74

TSR Test Suite Reduction. 60–62, 64–66, 80, 86, 87, 149

U.S.C. Code of Laws of the United States. 5

VSDM Versichertenstammdatenmanagement. 51

ZEH Zero Entry Hacking. 41, 43, 44

Bibliography

[1] *44 U.S. Code § 3542, Code of Laws of the United States*. URL: https://uscode.house.gov/browse/prelim@title44&edition=prelim. (visited on 22/04/2020).

[2] *ACT@Scale*. URL: https://webgate.ec.europa.eu/chafea_pdb/health/projects/709770/summary. (visited on 22/04/2020).

[3] O.H. Alhazmi, Y.K. Malaiya, and I. Ray. "Measuring, analyzing and predicting security vulnerabilities in software systems". In: *Computers & Security* 26.3 (2007), pp. 219 –228. ISSN: 0167-4048. DOI: 10.1016/j.cose.2006.10.002.

[4] P. Alpar et al. *Anwendungsorientierte Wirtschaftsinformatik*. Springer Fachmedien Wiesbaden, 2016. DOI: 10.1007/978-3-658-14146-2.

[5] Y. Amannejad et al. "A Search-Based Approach for Cost-Effective Software Test Automation Decision Support and an Industrial Case Study". In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. 2014, pp. 302–311. DOI: 10.1109/ICSTW.2014.34.

[6] K. Archie et al. "Test Automation Systems". Pat. 5,021,997. June 1999.

[7] B. Ballad, T. Ballad, and E. Banks. *Access Control, Authentication, and Public Key Infrastructure*. Jones & Bartlett Learning, 2010.

[8] M. Bishop. *Introduction to Computer Security*. Addison-Wesley Professional, 2006. ISBN: 0321247442.

[9] B. Blakley, E. McDermott, and D. Geer. "Information Security is Information Risk Management". In: *Proceedings of the 2001 Workshop on New Security Paradigms*. NSPW '01. Cloudcroft, New Mexico: ACM, 2001, pp. 97–104. ISBN: 1-58113-457-6. DOI: 10.1145/508171.508187.

[10] P. Bourque and R. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. 3rd ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014. ISBN: 9780769551661.

[11] *BSI-Standard 100-2 – IT-Grundschutz-Vorgehensweise*. Tech. rep. Bundesamt für Sicherheit in der Informationstechnik (BSI), 2008. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/BSIStandards/standard_100-2_e_pdf.pdf. (visited on 22/04/2020).

[12] *Bundesgesetz, mit dem das Bundes-Verfassungsgesetz geändert, das Datenschutzgesetz erlassen und das Datenschutzgesetz 2000 aufgehoben wird (Datenschutz- Anpassungsgesetz 2018).* Austrian law. 2018. URL: `https://www.ris.bka.gv.at/Dokumente/BgblAuth/BGBLA_2017_I_120/BGBLA_2017_I_120.pdfsig`. (visited on 22/04/2020).

[13] *Bundesgesetz, mit dem ein Telekommunikationsgesetz erlassen wird (Telekommunikationsgesetz 2003 – TKG 2003).* Austrian law. 2003. URL: `https://www.ris.bka.gv.at/Dokumente/BgblPdf/2003_70_1/2003_70_1.pdf`. (visited on 22/04/2020).

[14] *Bundesgesetz über den Schutz personenbezogener Daten (Datenschutzgesetz 2000 – DSG 2000).* Austrian law. 2000. URL: `https://www.ris.bka.gv.at/Dokumente/BgblPdf/1999_165_1/1999_165_1.pdf`. (visited on 22/04/2020).

[15] *Bundesgesetz vom 9. September 1955 über die Allgemeine Sozialversicherung (Allgemeines Sozialversicherungsgesetz - ASVG).* Austrian law. 1955. URL: `https://www.ris.bka.gv.at/Dokumente/BgblPdf/1955_189_0/1955_189_0.pdf`. (visited on 22/04/2020).

[16] T. Chen and M. Lau. "A simulation study on some heuristics for test suite reduction". In: *Information and Software Technology* 40.13 (1998), pp. 777 –787. ISSN: 0950-5849. DOI: `10.1016/S0950-5849(98)00094-9`.

[17] E. Collins et al. "An Industrial Experience on the Application of Distributed Testing in an Agile Software Development Environment". In: *2012 IEEE Seventh International Conference on Global Software Engineering.* 2012, pp. 190–194. DOI: `10.1109/ICGSE.2012.40`.

[18] *Committee on National Security Systems Instruction (CNSSI) No. 4009, Committee on National Security Systems (CNSS) Glossary.* 2015.

[19] *Common Criteria for Information Technology Security Evaluation, Version 3.1, Revision 5.* Standard. Geneva, CH: International Organization for Standardization, Apr. 2017.

[20] S. Darwall. *Philosophical Ethics: An Historical And Contemporary Introduction.* Routledge, 2018.

[21] J. Dick, E. Hull, and K. Jackson. *Requirements Engineering.* Springer International Publishing, 2017. DOI: `10.1007/978-3-319-61073-3`.

[22] W. Dickinson, D. Leon, and A. Fodgurski. "Finding failures by cluster analysis of execution profiles". In: *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001.* 2001, pp. 339–348. DOI: `10.1109/ICSE.2001.919107`.

[23] P. Drucker. "Managing for Business Effectiveness". In: *Harvard Business review* (1963), pp. 53–60.

[24] E. Dustin, T. Garrett, and B. Gauf. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Addison-Wesley Professional, 2009. ISBN: 978-0-321-58051-1.

[25] C. Eckert. *IT-Sicherheit - Konzepte, Verfahren, Protokolle (5. Aufl.)*. Oldenbourg, 2008, pp. I–XIV, 1–925. ISBN: 978-3-486-58270-3.

[26] *Elektronische Gesundheitsakte - ELGA*. URL: `https://www.elga.gv.at`. (visited on 22/04/2020).

[27] P. Engebretson. *The Basics of Hacking and Penetration Testing*. Elsevier, 2013. DOI: `10.1016/c2013-0-00019-9`.

[28] G. Eysenbach. "What is e-health?" In: *J Med Internet Res* 3.2 (2001), e20. DOI: `10.2196/jmir.3.2.e20`.

[29] Y. Fazlalizadeh et al. "Prioritizing test cases for resource constraint environments using historical test case performance data". In: *2009 2nd IEEE International Conference on Computer Science and Information Technology*. 2009, pp. 190–195. DOI: `10.1109/ICCSIT.2009.5234968`.

[30] J. Fieser. *Ethics*. URL: `http://www.iep.utm.edu/ethics`. (visited on 22/04/2020).

[31] *Flash Eurobarometer 404: European Citizen's digital health literacy*. Tech. rep. European Commission, 2014.

[32] V. Garousi and F. Elberzhager. "Test Automation: Not Just for Test Execution". In: *IEEE Software* 34.2 (2017), pp. 90–96. ISSN: 0740-7459. DOI: `10.1109/MS.2017.34`.

[33] D. Geer. "Are Companies Actually Using Secure Development Life Cycles?" In: *Computer* 43.6 (2010), pp. 12–16. ISSN: 0018-9162. DOI: `10.1109/MC.2010.159`.

[34] D. Geer and J. Harthorne. "Penetration testing: a duet". In: *18th Annual Computer Security Applications Conference, 2002. Proceedings*. 2002, pp. 185–195. DOI: `10.1109/CSAC.2002.1176290`.

[35] Gesellschaft für Telematikanwendungen der Gesundheitskarte mbH Gematik. *Einführung der Gesundheitskarte - Gesamtarchitektur*. Mar. 18, 2008. URL: `https://fachportal.gematik.de/fileadmin/user_upload/fachportal/files/Spezifikationen/Basis-Rollout/Architektur_und_uebergreifende_Dokumente/gematik_GA_Gesamtarchitektur_V1_3_0.pdf`. (visited on 22/04/2020).

[36] Gesellschaft für Telematikanwendungen der Gesundheitskarte mbH Gematik. *Informationsblatt: Technische Ausstattung einer medizinischen Einrichtung*. Oct. 1, 2017. URL: `https://fachportal.gematik.de/fileadmin/user_upload/gematik/files/OPB-Infomaterialien/gem_2017-12-IB-TAME_technische_ausstattung_online.pdf`. (visited on 22/04/2020).

[37] *Gesetz für sichere Kommunikation und Anwendungen im Gesundheitswesen sowie zur Änderung weiterer Gesetze.* German law. 2015. URL: `https://www.bgbl.de/xaver/bgbl/start.xav`. (visited on 22/04/2020).

[38] M. Harman and P. McMinn. "A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search". In: *IEEE Transactions on Software Engineering* 36.2 (2010), pp. 226–247. ISSN: 0098-5589. DOI: `10.1109/TSE.2009.71`.

[39] M. J. Harrold, R. Gupta, and M. L. Soffa. "A Methodology for Controlling the Size of a Test Suite". In: *ACM Trans. Softw. Eng. Methodol.* 2.3 (July 1993), pp. 270–285. ISSN: 1049-331X. DOI: `10.1145/152388.152391`.

[40] K. M. Henry. *Penetration Testing: Protecting Networks and Systems.* IT Governance Publishing, 2012.

[41] M. Howard and S. Lipner. *The Security Development Lifecycle.* Redmond, WA, USA: Microsoft Press, 2006. ISBN: 0735622140.

[42] *IEEE Standard Glossary of Software Engineering Terminology.* Tech. rep. 1990, pp. 1–84. DOI: `10.1109/IEEESTD.1990.101064`.

[43] German Federal Office for Information Security. *Study: A Penetration Testing Model.* 2003.

[44] *ISO/IEC 27000:2017 – Information security management systems – Overview and vocabulary.* Standard. International Organization for Standardization, 2017.

[45] *ISO/IEC 27000:2018 – Information security management systems – Overview and vocabulary.* Standard. International Organization for Standardization, 2018.

[46] P. Jaccard. "Lois de distribution florale dans la zone alpine". In: *Bulletin de la Société Vaudoise des Sciences Naturelles* 28.144 (1902).

[47] N. F. Johnson. *Simply complexity : a clear guide to complexity theory.* Oneworld, London, 2009.

[48] A. Kerkhoffs. "La Cryptographie Militaire". In: *Journal des Sciences Militaires* 9 (1883), pp. 161–191.

[49] A. Khalilian, M. A. Azgomi, and Y. Fazlalizadeh. "An improved method for test case prioritization by incorporating historical test case data". In: *Science of Computer Programming* 78.1 (2012). Special Section: Formal Aspects of Component Software (FACS'09), pp. 93 –116. ISSN: 0167-6423. DOI: `10.1016/j.scico.2012.01.006`.

[50] J. Kim and A. Porter. "A History-based Test Prioritization Technique for Regression Testing in Resource Constrained Environments". In: *Proceedings of the 24th International Conference on Software Engineering.* ICSE '02. Orlando, Florida: ACM, 2002, pp. 119–129. ISBN: 1-58113-472-X. DOI: `10.1145/581339.581357`.

[51] H. Kumar. *Learning Nessus for Penetration Testing*. Packt Publishing, 2014. ISBN: 978-1-78355-099-9.

[52] Y. Ledru et al. “Prioritizing test cases with string distances”. In: *Automated Software Engineering* 19.1 (2012), pp. 65–95. ISSN: 1573-7535. DOI: 10.1007/s10515-011-0093-0.

[53] V. I. Levenshtein. “Binary codes capable of correcting deletions, insertions and reversals”. In: *Soviet Physics Doklady* 10.8 (1966).

[54] S. McConnell. *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004. ISBN: 0735619670, 9780735619678.

[55] G. McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006. ISBN: 0321356705.

[56] C. Meudec. “ATGen: automatic test data generation using constraint logic programming and symbolic execution”. In: *Software Testing, Verification and Reliability* 11.2 (2001), pp. 81–96. DOI: 10.1002/stvr.225.

[57] J. C. Miller and C. J. Maloney. “Systematic Mistake Analysis of Digital Computer Programs”. In: *Commun. ACM* 6.2 (Feb. 1963), pp. 58–63. ISSN: 0001-0782. DOI: 10.1145/366246.366248.

[58] G. J. Myers, T. Badgett, and Sandler C., eds. *The Art of Software Testing*. John Wiley & Sons, Inc., 2012. DOI: 10.1002/9781119202486.

[59] Neustar. *Worldwide DDoS Attacks & Cyber Insights Research Report*. 2017.

[60] S. Nora and A. Minc. *L’informatisation de la société: rapport à M. le Président de la République*. La Documentation Française, 1978.

[61] H. Oh et al. “What Is eHealth (3): A Systematic Review of Published Definitions”. In: *J Med Internet Res* 7.1 (2005), e1. DOI: 10.2196/jmir.7.1.e1.

[62] Y. Onn et al. *Privacy in the Digital Environment*. Tech. rep. The Haifa Center of Law and Technology Publication Series, 2005.

[63] *ÖNORM EN ISO 9000:2015 Quality Management Systems - Fundamentals and vocabulary*. Tech. rep. International Organization for Standardization, translated by Austrian Standards Institute, 2015.

[64] *ÖNORM ISO 31000:2009 Grundsätze und Richtlinien*. Tech. rep. International Organization for Standardization, translated by Austrian Standards Institute, 2010.

[65] *ONR 49000:2014 Risikomanagement für Organisationen und Systeme*. Tech. rep. Austrian Standards Institute, 2014.

[66] *OWASP - Risk Rating Methodology*. URL: https://wiki.owasp.org/index.php/OWASP_Risk_Rating_Methodology. (visited on 22/04/2020).

[67] European Parliament and Council. “Regulation (EU) 2016/679 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)”. In: *Official Journal of the European Union, L 119* (2016).

[68] R. Patton. *Software Testing*. Sams Pub., 2006. ISBN: 9780672327988.

[69] T. Peng, C. Leckie, and K. Ramamohanarao. “Survey of Network-Based Defense Mechanisms Countering the DoS and DDoS Problems”. In: *ACM Comput. Surv.* 39.1 (Apr. 2007). ISSN: 0360-0300. DOI: 10.1145/1216370.1216373.

[70] K. P. Pfeiffer. “Ist-Zustand und Perspektive von e-Health in Österreich und International – ein Überblick”. In: *Wiener Medizinische Wochenschrift* 161.13 (2011), pp. 334–340. ISSN: 1563-258X. DOI: 10.1007/s10354-011-0008-5.

[71] C. Pfleeger and S. Pfleeger. *Security in Computing*. 4th ed. Prentice Hall, 2007. ISBN: 9780132390774.

[72] D. Pierce, A. Jones, and M. Warren. “Penetration Testing Professional Ethics: a conceptual model and taxonomy”. In: *Australasian Journal of Information Systems* 13.2 (2006). DOI: 10.3127/ajis.v13i2.52.

[73] *Recommendation for an Austrian E-Health Strategy*. Tech. rep. Task Force 1, Austrian E-Health Inititive (EHI), 2007.

[74] S. U. Rehman Khan et al. “A Systematic Review on Test Suite Reduction: Approaches, Experiment’s Quality Evaluation, and Guidelines”. In: *IEEE Access* 6 (2018), pp. 11816–11841. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2809600.

[75] R. Ross, M. McEvilley, and J. C. Oren. *NIST Special Publications 800-160 Systems Security Engineering*. Tech. rep. National Institute of Standards and Technology, 2016.

[76] C. Rossow. “Amplification Hell: Revisiting Network Protocols for DDoS Abuse.” In: *NDSS*. 2014.

[77] G. Rothermel, R. H. Untch, and M. J. Harrold. “Prioritizing test cases for regression testing”. In: *IEEE Transactions on Software Engineering* 27.10 (2001), pp. 929–948. ISSN: 0098-5589. DOI: 10.1109/32.962562.

[78] G. Rothermel et al. “Empirical studies of test-suite reduction”. In: *Software Testing, Verification and Reliability* 12.4 (2002), pp. 219–249. DOI: 10.1002/stvr.256.

[79] K. Scarfone et al. *NIST Special Publications 800-115 Technical Guide to Information Security Testing and Assessment*. Tech. rep. National Institute of Standards and Technology, 2008.

[80] C. Schanes et al. "Problem Space and Special Characteristics of Security Testing in Live and Operational Environments of Large Systems Exemplified by a Nationwide IT Infrastructure". In: *2009 First International Conference on Advances in System Testing and Validation Lifecycle*. 2009, pp. 161–166. DOI: 10.1109/VALID.2009.24.

[81] A. Schopenhauer. *Die beiden Grundprobleme der Ethik, behandelt in zwei akademischen Preisschriften*. F. A. Brockhaus, 1860.

[82] *SeleniumHQ Browser Automation*. URL: https://www.seleniumhq.org. (visited on 22/04/2020).

[83] Y. Singh. *Software Testing*. Cambridge University Press, 2011. DOI: 10.1017/CBO9781139196185.

[84] I. Sommerville et al. "Large-scale Complex IT Systems". In: *Commun. ACM* 55.7 (July 2012), pp. 71–77. ISSN: 0001-0782. DOI: 10.1145/2209249.2209268.

[85] *Sozialgesetzbuch (SGB) Fünftes Buch (V), § 291 Elektronische Gesundheitskarte als Versicherungsnachweis*. German law.

[86] A. Spillner et al. *Software Testing Practice: Test Management*. Rocky Nook, 2007.

[87] W. Stallings and L. Brown. *Computer Security: Principles and Practice*. 3rd. 2015.

[88] *Statistik über Versicherte, gegliedert nach Status, Alter, Wohnort und Kassenart*. German Federal Ministry of Health. 2019. URL: https://www.bundesgesundheitsministerium.de/fileadmin/Dateien/3_Downloads/Statistiken/GKV/Mitglieder_Versicherte/KM6_2019.xlsx. (visited on 22/04/2020).

[89] D. Stevens. "Appendix A: Examples of eHealth Solutions Featured at the Workshop". eng. In: *The Promises and Perils of Digital Strategies in Achieving Health Equity: Workshop Summary*. National Academies Press, 2016, pp. 49–56. ISBN: 0309438918.

[90] *Telematikinfrastruktur: Informationen zum Anschluss der Praxen, zur technischen Ausstattung und zur Finanzierung*. Tech. rep. Kassenärztliche Bundesvereinigung, 2019.

[91] *Tenable - Nessus*. URL: https://www.tenable.com. (visited on 22/04/2020).

[92] *The Penetration Testing Execution Standard (PTES)*. The PTES Team. 2017. URL: http://pentest-standard.org/. (visited on 22/04/2020).

[93] H. Thompson. "Why security testing is hard". In: *IEEE Security Privacy* 1.4 (2003), pp. 83–86. ISSN: 1558-4046. DOI: 10.1109/MSECP.2003.1219078.

[94] *Verordnung der Bundesministerin für Verkehr, Innovation und Technologie betreffend die Datensicherheit (Datensicherheitsverordnung TKG-DSVO), Österreich*. 2011.

[95] K. Wiklund and M. Wiklund. “The Next Level of Test Automation: What About the Users?” In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2018, pp. 159–162. DOI: 10.1109/ICSTW.2018.00045.

[96] P. Wilson, C. Leitner, and A. . Moussalli. *Mapping the Potential of eHealth: Empowering the Citizen through eHealth Tools and Services. Research Report presented at the eHealth Conference, Cork, Ireland, 5-6 May 2004*. ISBN: 9067791881. European Institute of Public Administration, Maastricht, 2004, p. 52.

[97] W. E. Wong et al. “A study of effective regression testing in practice”. In: *Proceedings The Eighth International Symposium on Software Reliability Engineering*. 1997, pp. 264–274. DOI: 10.1109/ISSRE.1997.630875.

[98] D. Xu et al. “Automated Security Test Generation with Formal Threat Models”. In: *IEEE Transactions on Dependable and Secure Computing* 9.4 (2012), pp. 526–540. ISSN: 1545-5971. DOI: 10.1109/TDSC.2012.24.