FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Analyse und Optimierung von Sicherheitstestergebnissen durch Anwendung von Data-Mining-Methoden

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Herbert Brunner
Matrikelnummer 0627669

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, November 28, 2011 _____ _____

(Unterschrift Verfasser) (Unterschrift Betreuung)

Analyse und Optimierung von Sicherheitstestergebnissen durch Anwendung von Data-Mining-Methoden

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Herbert Brunner
Matrikelnummer 0627669

ausgeführt am
Institut für Rechnergestützte Automation
Forschungsgruppe Industrial Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, November 28, 2011

Eidesstattliche Erklärung

Herbert Brunner
Gartengasse 17, 1050 Wien

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

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

(Ort, Datum) (Unterschrift Verfasser)

Abstract

Nowadays the internet and computers are two driving forces. Due to this fact more and more software is developed and released in a shorter amount of time. Software may have bugs that can compromise systems to work in the proper way. These defects can be used by attackers to exploit a software and for instance steal sensitive data. To prevent software from beeing attacked it needs to be tested properly. Security tests offer a possibility to evaluate software security, for example by the use of automatic tests, e. g., fuzzing. The problem with automated security tests is the number of false positive test results which have to be checked manually by the tester.
The aim of this thesis is to analyze these results with data mining methods, which consist of association mining, classification, cluster analysis (k-means) and text mining. These strategies are used to verify if the test case results produced by the fuzzer are correct or not and therefore reduce the number of false positive detected errors.

Keywords

data mining, cluster analysis, text mining, association mining, classification, fuzzing, software security, software testing

Kurzfassung

Die Anzahl an neuen Software-Produkten ist heute kaum mehr zu überblicken, wobei Software häufig fehlerhaft auf den Markt gebracht wird. Diese Fehler können die Funktion von Software beeinträchtigen, da potenzielle Angreifer dadurch Software kompromittieren und Schaden an sensiblen Daten anrichten können. Deshalb muss Software auf Sicherheitslücken getestet werden, zum Beispiel durch die Verwendung von automatisierten Tests mit der Hilfe von Fuzzern. Das Problem bei automatisierten Sicherheitstests ist die Anzahl an falsch positiven Testergebnissen, welche nachträglich vom Tester überprüft werden müssen.
Im Zuge dieser Arbeit werden Sicherheitstestergebnisse mit Data-Mining-Methoden analysiert. Diese sind Association-Mining, Klassifikation, Clusteranalyse (K-Means) und Text-Mining.
Durch Anwendung dieser Data-Mining-Methoden wird die Anzahl der falsch erkannten Fehler des Testwerkzeugs reduziert und die Arbeit der Sicherheitstester vereinfacht.

Schlüsselwörter

Data-Mining, Clusteranalyse, Text-Mining, Assoziationsregeln, Klassifikation, Fuzzing, Softwaresicherheit, Softwaretest

Inhaltsverzeichnis

Inhaltsverzeichnis v

Abbildungsverzeichnis vii

Tabellenverzeichnis viii

Listings x

1 Einleitung 1
- 1.1 Problemstellung und Motivation . . . 1
- 1.2 Zielsetzung . . . 1
- 1.3 Struktur der Arbeit . . . 2

2 Einführung in Softwaresicherheit 3
- 2.1 Softwaresicherheit . . . 3
- 2.2 Sicherheitsanforderungen vs. sichere Anforderungen . . . 4
- 2.3 Sicherheitsanforderungen und Missbrauchfälle . . . 6
- 2.4 Design und Risikoanalyse . . . 7
- 2.5 Planung von Sicherheitstests . . . 7
- 2.6 Implementierung und Codereview . . . 7
- 2.7 Durchführung von Penetrationtests . . . 7
- 2.8 Feedback und Inbetriebnahme . . . 8

3 Softwaretesten 9
- 3.1 Aspekte beim Softwaretesten . . . 9
  - 3.1.1 Funktionstest . . . 9
  - 3.1.2 Performancetest . . . 11
  - 3.1.3 Interfacetest . . . 11
  - 3.1.4 Smoketest . . . 12
- 3.2 Softwaretestprozess . . . 12
  - 3.2.1 V-Modell . . . 12
  - 3.2.2 Softwarefehler . . . 13
  - 3.2.3 Softwareteststufen . . . 14
- 3.3 Verifikation von Softwaresicherheit . . . 17
  - 3.3.1 Testautomatisierung mittels Fuzzing . . . 18

4 Verarbeitung von Datenmengen mittels Data-Mining 22
- 4.1 Data-MiningProzess . . . 22
- 4.2 Daten und ihre Herkunft . . . 23
  - 4.2.1 Relationale Datenbanken . . . 24
  - 4.2.2 Data-Warehouse . . . 24
- 4.3 Muster in Daten . . . 24
- 4.4 Wissensvorbereitung . . . 25
  - 4.4.1 Konzepte, Instanzen und Attribute . . . 25
  - 4.4.2 Attribute und ihre Ausprägungen . . . 26
  - 4.4.3 Datenpräparierung . . . 27

4.5 Wissensdarstellung 28
4.5.1 Tabellen 28
4.5.2 Bäume 29
4.5.3 Regeln 29
4.5.4 Cluster 31

5 Fallbeispiel Fuzzing-Werkzeug 32
5.1 Architektur des Fuzzers 32
5.2 Datenbankstruktur des Fuzzing-Werkzeugs 34
5.3 Sicherheitsbedrohnungen am Fallbeispiel des Fuzzing-Werkzeugs 40

6 Erkennung von Sicherheitsfehlern durch Data-Mining 47
6.1 Assoziationsregeln 47
6.1.1 Einfacher Algorithmus 47
6.1.2 Apriori-Algorithmus 49
6.2 Klassifikation 50
6.2.1 Wahl der Klassifikationsattribute 51
6.2.2 Klassifikation mit Entscheidungsbäumen 52
6.2.3 Klassifikation nach der Bayes'schen Methode 54
6.3 Clusteranalyse 56
6.3.1 Distanzmaß 56
6.3.2 Arten der Clusteranalyse 56
6.3.3 K-Means-Methode 57
6.4 Text-Mining 60
6.4.1 Textdarstellung im Rahmen von Text-Mining 60
6.4.2 Selektionsmechanismen für Wörter aus Texten 61
6.4.3 Stoppwort-Entfernung und Vereinheitlichung der Schreibweise 61
6.4.4 Wortstammreduktion 61
6.4.5 Filterung mittels Vector-Space-Model 62
6.4.6 Vektornormalisierung und -normierung 62
6.4.7 Textvergleich mit Hilfe des Vector-Space-Models 63

7 Anwendung von Data-Mining am Beispiel eines Fuzzing-Werkzeugs 65
7.1 Vorgehen der Analyse 65
7.2 Bereiningung und Transformation der Daten 65
7.3 Analyse des Datenbestandes mittels Data-Mining 66
7.4 Evaluation der Analyseergebnisse 76

8 Zusammenfassung und Ausblick 78

Literatur 79

Abbildungsverzeichnis

2.1 Berührungspunkte zwischen Softwareentwicklung und Softwaresicherheit [vgl. [54]] 6

3.1 Schematische Darstellung eines Black-Box-Tests [vgl. [1]] 10
3.2 Schematische Darstellung eines White-Box-Tests [vgl. [1]] 11
3.3 Softwaretestprozess mittels V-Modell [vgl. [46]] 13
3.4 Schematische Darstellung eines Modultests [vgl. [1]] 15
3.5 Schematische Darstellung eines Integrationstests [vgl. [38]] 16
3.6 Schematische Darstellung eines Systemtests [vgl. [38]] 17
3.7 Schematische Darstellung der Durchführung des Fuzzing-Prozesses [vgl. [71]] . . 20

5.1 Schematische Darstellung der Architektur des Fuzzing-Werkzeugs [vgl. [67]] . . . 33
5.2 Schematische Darstellung der StateMachine-Phasen des Fuzzers beim Testen von SIP-basierten Softphones [vgl. [67]] 34
5.3 Exemplarisches Datenmodell des Fuzzing-Werkzeugs[vgl. [67]] 39
5.4 Schematischer Ablauf einer SQL-Injection während eines Fuzzing-Testlaufs [vgl. [67]] 41
5.5 Schematischer Ablauf einer Character-Injection durch den Fuzzer während eines Testlaufs [vgl. [67]] 43
5.6 Schematischer Ablauf einer Command-Injection während eines Fuzzing-Testlaufs [vgl. [67]] 45

6.1 Graphische Darstellung des Entscheidungsbaums nach der Klassifikation von Softwarefehlern [vgl. [32, 35, 10]] 54

7.1 Schematische Darstellung des Analyseprozesses mittels Apriori-Algorithmus [vgl. [32]] 67
7.2 Schematische Darstellung des Analyseprozesses von Ergebnissen des FileReader-Analyzers 70
7.3 Schematische Darstellung des Analyseprozesses von Ergebnissen des Response-Analyzers 71
7.4 Schematische Darstellung des Analyseprozesses von Ergebnissen des ResponseTimeAnalyzers 72
7.5 Schematische Darstellung des ganzheitlichen Analyseprozesses der einzelnen Analyzerergebnisse 74

Tabellenverzeichnis

5.1 Beschreibung der Tabelle FILEREADER_ANALYZER [vgl. [67]] 35
5.2 Beschreibung der Tabelle FILEREADER_ANALYZER_KEYWORDS [vgl. [67]] . 35
5.3 Beschreibung der Tabelle RESPONSE_ANALYZER [vgl. [67]] 36
5.4 Beschreibung der Tabelle RESPONSE_CLASSSIFICATION_ANALYZER [vgl. [67]] 36
5.5 Beschreibung der Tabelle RESPONSETIME_ANALYZER [vgl. [67]] 37
5.6 Beschreibung der Tabelle CPU_LOAD_ANALYZER [vgl. [67]] 37
5.7 Beschreibung der Tabelle MEMORY_LOAD_ANALYZER [vgl. [67]] 37
5.8 Beschreibung der Tabelle TCPPORT_ANALYZER [vgl. [67]] 38
5.9 Beschreibung der Tabelle ANALYZER_RESULT [vgl. [67]] 38
5.10 Beschreibung der Tabelle TESTCASE_VARIATION [vgl. [67]] 38
5.11 Hauptspeicheransicht eines Rechners vor der Erzeugung eines Pufferüberlaufs [vgl. [51]] 42
5.12 Hauptspeicheransicht eines Rechners nach der Erzeugung eines Pufferüberlaufs [vgl. [51]] 43

6.1 Ausgangssituation bei der Generierung von Assoziationsregeln durch den einfachen Algorithmus [vgl. [32]] 48
6.2 Kombination von Testfallfehlermeldungen zur Generierung von Assoziationsregeln durch den einfachen Algorithmus [vgl. [32]] 48
6.3 Ausgangssituation zum Finden von Assoziationen zwischen Softwarefehlermeldungen mittels Apriori-Algorithmus [vgl. [32]] 49
6.4 Generierung des Kandidatensets C2 zum Finden von Assoziationen zwischen Softwarefehlern mittels Apriori-Algorithmus [vgl. [32]] 50
6.5 Ausgangssituation bei der Klassifikation von Sicherheitstestergebnissen mittels Entscheidungsbaum [vgl. [32]] 52
6.6 Ausgangssituation bei der Clusteranalyse von Sicherheitstestergebnissen nach der K-Means-Methode [vgl. [10]] 58
6.7 Initiale Mittelwerte bei der Clusteranalyse von Sicherheitstestergebnissen nach der K-Means-Methode [vgl. [10]] 58
6.8 Clustereinteilung von Sicherheitstestergebnissen mittels K-Means-Methode nach der ersten Iteration [vgl. [10]] 58
6.9 Mittelwerte bei der Clusteranalyse von Sicherheitstestergebnissen mittels K-Means-Methode nach der ersten Iteration [vgl. [10]] 58
6.10 Clustereinteilung von Sicherheitstestergebnissen mittels K-Means-Methode nach der zweiten Iteration [vgl. [10]] 59
6.11 Finale Clusterung von Sicherheitstestergebnissen mittels K-Means-Methode [vgl. [10]] 59
6.12 Ausgangssituation bei der Normierung von Vektoren [vgl. [10]] 62

7.1 Schematische Darstellung der Tabelle TESTCASE_VARIATION vor der Durchführung von ETL [vgl. [32]] 66
7.2 Schematische Darstellung der Tabelle TESTCASE_VARIATION nach der Durchführung von ETL [vgl. [32]] 66
7.3 Schematische Vorbereitung der Ausgangstabelle zur Durchführung des Apriori-Algorithmus [vgl. [32]] 68

7.4 Schematische Ermittlung von aufgetretenen Fehlermeldungen gruppiert nach deren Häufigkeit zur Durchführung des Apriori-Algorithmus [vgl. [32]] 68
7.5 Schematische Ermittlung von Fehlermeldungskombinationen gruppiert nach deren Häufigkeit zur Durchführung des Apriori-Algorithmus [vgl. [32]] 69
7.6 Schematische Ermittlung der Häufigkeit pro aufgetretener Fehlermeldungen beim FileReaderAnalyzer zur Überprüfung von Auffälligkeiten 70
7.7 Schematische Ermittlung der Länge pro erhaltener Antwortnachricht beim ResponseAnalyzer zur Überprüfung von Auffälligkeiten 72
7.8 Schematische Ermittlung der Antwortzeit gruppiert nach der Häufigkeit beim ResponseTimeAnalyzer zur Überprüfung von Auffälligkeiten 73
7.9 Schematische Ermittlung der Dauer pro erhaltener Antwortnachricht beim ResponseTimeAnalyzer zur Überprüfung von Auffälligkeiten 73
7.10 Tabelle zur Überprüfung der gesamten Fehlerklassifikation durch den Fuzzer . . . 75
7.11 Schematische Tabelle mit den analysierten Testläufen sowie deren Testfallanzahl und falsch positiven Ergebnissen 76
7.12 Schematische Tabelle mit der Aufschlüsselung der falsch positiven Ergebnisse pro Testlauf und Analyzer 76

Listings

5.1 Exemplarisches Beispiel einer SQL-Abfrage [vgl. [51]] 40
5.2 Präparierte Abfrage zur Erzeugung einer SQL-Injection [vgl. [51]] 40
5.3 Darstellung einer SQL-Injection durch den Fuzzer in Form eines HTTP-Headers [vgl. [67]] 41
5.4 Darstellung einer Antwortnachricht des SUT auf eine SQL-Injection des Fuzzers in Form eines HTTP-Headers[vgl. [67]] 42
5.5 Darstellung einer Character-Injection durch den Fuzzer in Form eines SOAP-Requests [vgl. [67]] 44
5.6 Darstellung einer Antwortnachricht des SUT in Form einer SOAP-Response auf eine Character-Injection des Fuzzers [vgl. [67]] 44
5.7 Schematische Darstellung einer Command-Injection [vgl. [42]] 45
5.8 Darstellung einer Command-Injection durch den Fuzzer in Form eines HTTP-Headers [vgl. [67]] 46
5.9 Darstellung einer Antwortnachricht des SUT auf eine Command-Injection des Fuzzers in Form eines HTTP-Headers [vgl. [67]] 46
7.1 Schematische Darstellung einer Abfrage zur Überprüfung der gesamten durch den Fuzzer erstellten Fehlerklassifikation 75

1 Einleitung

1.1 Problemstellung und Motivation

In der heutigen Zeit verwenden immer mehr Menschen Software. Sie befindet sich beispielsweise in Flugzeugen oder Automobilen, aber auch in kleineren Geräten wie Personal Computer und deren Mikrochips. Dabei dient Software nicht nur zur Steuerung solcher Gegenstände, sondern sie interagiert auch mit dem Menschen und dessen Umwelt.
Viele Menschen verwenden Software hauptsächlich im Berufsleben zur Erledigung ihrer allfälligen Tätigkeiten, aber auch privat findet Software ihren Einsatz, beispielsweise in Zusammenhang mit dem Internet. Softwarebenutzer können über E-Mail, diverse Chatprogramme oder Foren ihr Leben und ihre Meinungen miteinander austauschen.
Bevor Software verwendet werden kann, muss sie zuerst konzipiert und entwickelt werden. Im Rahmen der Softwareentwicklung werden Softwaretests zur Überprüfung der Funktionalität und Qualität von Software eingesetzt. Software kann auf verschiedene Art und Weise getestet werden. Tests stehen jedoch nicht als Garant dafür, dass die getestete Software komplett fehlerfrei ist, sondern sagen nur aus, ob etwaige Softwarefehler im Zuge eines Testlaufs gefunden werden konnten.
Ein Aspekt bei der Entwicklung von Software stellt das Thema Sicherheit dar. Durch die steigende Vernetzung von Hardware beziehungsweise Software durch das Internet, wird dieser Bereich immer wichtiger. Menschen tendieren dazu mehr und mehr berufliche und persönliche Daten im Internet auf unterschiedlichen Seiten von sich preiszugeben. Je unsicherer jedoch Software im Umgang mit vertraulichen Daten ist, desto höher ist die Gefahr, dass potenzielle Angreifer diese Sicherheitslücken ausnützen und Schaden anrichten. Dieser Schaden kann sich beispielsweise auf den Verkauf von sensiblen Daten sowie Imageschädigung von Personen oder Unternehmen beziehen.
Sicherheitsfehler können bereits während der Konzeption von Software durch Designschwächen oder im Zuge der Entwicklung implementiert worden sein. Daher muss Software auf Schwachstellen mit Hilfe von entsprechenden Sicherheitstests untersucht und geprüft werden. Es gibt aktuell verschiedene Ansätze, um Sicherheitstests durchzuführen; einer davon stellt das so genannte Fuzzing dar. Im Rahmen von Fuzzing kann Software auf automatisierte Art und Weise mit einer Vielzahl von Testfällen auf Sicherheitsfehler getestet werden. Konkret wird in dieser Arbeit ein Fuzzer verwendet, welcher nach jedem Softwaretestlauf große Mengen an Testergebnisdaten generiert. Diese Daten beinhalten beispielsweise Informationen darüber, ob Sicherheitsfehler in der zu testenden Software gefunden werden konnten respektive um welche Sicherheitsfehler es sich dabei gehandelt hat. Die Problematik ist, dass der Fuzzer neben positiven auch falsch positive und falsch negative Testfallergebnisse produziert. Im Zuge dieser Arbeit wird daher eine gesamtheitliche Analyse der generierten Ergebnisse des Fuzzers durchgeführt, welche die zu Grunde liegenden Informationen evaluiert und Aussagen darüber trifft.

1.2 Zielsetzung

Im Rahmen dieser Arbeit werden die Ergebnisse von Sicherheitstests analysiert und optimiert. Die Optimierungsarbeiten umfassen einerseits die Reduktion der falsch positiven Bewertungsergebnisse des Fuzzers, andererseits wird versucht neue Erkenntnis über Sicherheitsfehler aus den Testergebnisdaten zu gewinnen.

Zu diesem Zweck werden Methoden aus dem Bereich Data-Mining verwendet. Data-Mining beschäftigt sich mit der Analyse und Evaluation von Information in (großen) Datenmengen. Ziel ist es mittels Data-Mining Fehlermuster in den Daten zu erkennen. Es werden Abhängigkeiten und Korrelationen innerhalb von Testergebnissen gesucht. Außerdem werden die Testergebnisse auf ähnliche, charakteristische Eigenschaften untersucht. Auf Grund dieser Erkenntnis wird untersucht, ob sich die Testfälle in entsprechende Kategorien einteilen lassen. Es wird ebenso versucht Fehler nach ihrer Kritikalität zu klassifizieren. Des Weiteren wird evaluiert, ob manche Fehler häufiger oder weniger häufig als andere aufgetreten sind.

1.3 Struktur der Arbeit

Der Aufbau dieser Arbeit gliedert sich wie folgt: Im Kapitel 2 werden grundlegende Begriffe aus dem Bereich Softwaresicherheit erläutert. Kapitel 3 beschreibt das Thema Softwaretesten allgemein und speziell bezogen auf den Bereich Sicherheitstests mit Hilfe von Fuzzing. Im Kapitel 4 wird Data-Mining allgemein vorgestellt. Kapitel 5 beschreibt Fuzzing speziell in Hinblick auf den in der Arbeit verwendeten Fuzzer. Im Kapitel 6 wird Data-Mining in Verbindung mit Softwaresicherheitsfehlern gebracht. Es werden schematische Beispiele gegeben, wie Data-Mining für die Analyse und Optimierung von Sicherheitstestergebnissen verwendet werden kann. Das Kapitel 7 beinhaltet die Evaluation der Sicherheitstestergebnisse aus dem vorangegangenen Analyseprozess bezogen auf den konkreten Fuzzer. Im Kapitel 8 wird der Inhalt dieser Arbeit nochmals zusammengefasst und anschließend ein Ausblick gegeben.

2 Einführung in Softwaresicherheit

In diesem Kapitel werden Begrifflichkeiten zum Thema Sicherheit und Software erklärt. Diese beziehen sich auf die sichere Entwicklung und vor allem auf das Sicherheitstesten von Software. Dabei wird der sichere Entwurf beziehungsweise die sichere Implementierung von Software sowie die anschließende Verifikation mit Hilfe von Sicherheitstests erläutert.

2.1 Softwaresicherheit

"Software ist überall." [McGraw, 2006, [54]] Durch die immer schneller werdende Softwareentwicklung wird der Markt laufend mit neuen Softwareprodukten förmlich überflutet. Die zunehmende Computerisierung der Menschheit schafft eine gewisse Softwareabhängigkeit. Der Wert von Software beruht dabei allerdings nicht nur auf der ordnungsgemäßen Arbeitsverrichtung von dieser, sondern zunehmend auch auf der sachgerechten Ausführung der Arbeit unter risikoreichen und kritischen Aspekten.
Ein zentrales Problem in diesem Bereich stellt daher das Thema Softwaresicherheit dar. [vgl. [3]] In der Vergangenheit kam es immer wieder zu Unfällen, welche durch mangelnde Softwaresicherheit passiert sind. Hier seien der Toyota Prius Bug oder Atomubootunfälle in Russland erwähnt. [vgl. [47]]
In den Anfängen der IT wurde dem Bereich Softwaresicherheit mitunter nicht immer die nötige Beachtung geschenkt. Software wurde damals nur für einen beschränkten Benutzerkreis konzipiert und diesem zugänglich gemacht. Heutzutage hat sich dieser Kreis von Benutzern durch das Internet sehr vergrößert. Es werden ständig neue Softwareplattformen und –systeme im Internet angeboten und einer breiten Masse von Anwendern zur Verfügung gestellt. Die Interaktion zwischen verschiedenen Benutzern wird somit über das Internet ermöglicht und gefördert. Die Kehrseite der Medaille ist, dass Software dadurch potenzielles Objekt für Angriffe werden kann. Diese Problematik wird verursacht durch die Komplexität von Software, deren Erweiterbarkeit (mittels Updates und Gadgets) und die bereits erwähnte Konnektivität per Internet. [vgl. [3]] Es gilt daher Software vor Angriffen möglichst sicher zu machen.

Softwaresicherheit baut dabei auf drei Grundsteinen auf [vgl. [9]]:

- Vertraulichkeit
- Integrität
- Verfügbarkeit

Die Vertraulichkeit von Software basiert auf verschiedenen Zugriffskontrollmechanismen. Nicht alle Benutzer einer Software haben beispielsweise auf alle Daten des Systems Zugriff. Außerdem kann die Vertraulichkeit von Daten mittels Kryptographie unterstützt werden. Auf diese Weise können sensible Daten verschlüsselt und vor nicht befugten Personen verborgen werden.
Integrität zielt auf den Inhalt der zu schützenden Daten und deren Glaubwürdigkeit ab. Dateninhalte dürfen nicht unautorisiert und willkürlich verändert werden.
Die Verfügbarkeit von Daten ist ein ebenso wichtiger Aspekt, welcher gewährleistet werden muss. Ein System darf nicht kompromittiert werden, sodass Benutzern der Zugang zu ihren Daten beispielsweise durch einen Hacker-Angriff verwehrt wird. [vgl. [9, 3, 54, 47]]

2.2 Sicherheitsanforderungen vs. sichere Anforderungen

In der englischen Literatur lassen sich mehrere Definitionen zum Terminus Sicherheit finden, nämlich: ”Security” und ”Safety”. Der Begriff ”Safety” bezeichnet dabei die Abwendung von Gefahr durch das Eintreten von unwillkürlichen Umständen. Das können sein: Gefahren bezogen auf die Gesundheit oder das Umfeld sowie die Eigenschaften einer Sache. ”Security” wiederum definiert den Schutz einer Sache, der Gesundheit, des Lebens usw. durch das willkürlich herbeigeführte Eintreten von Gefahren. [vgl. [3, 26]]
Diese Definitionen können auch auf die Softwareebene übertragen werden. In diesem Zusammenhang seien Hazards erwähnt. Sie kommen durch Unfälle (z. B. Brand in einem Rechenzentrum, Erdbeben usw.) zustande und beeinträchtigen auf unwillkürliche Weise die Sicherheit von Software. Exploits (z. B. Hacking eines Datenbankrechners oder einer Internetseite) hingegen stellen einen willkürlichen Angriff auf die Softwaresicherheit dar. [vgl. [47]]
Um Softwaresicherheit implementieren zu können, bedarf es daher einer vorgelagerten Analysephase, welche zur Erhebung von Sicherheitsanforderungen dient. Dabei können diese in zwei Kategorien eingeteilt werden, zum einen in funktionale Sicherheitsanforderungen und zum anderen in sicherheitsgewährleistende Anforderungen.
Funktionale Sicherheitsanforderungen definieren beispielsweise wie sich bestimmte Benutzer mit zugewiesenen Rollen bei einem Softwaresystem anzumelden haben oder wie sich die Verwaltung und Verteilung von kryptographischen Schlüsseln in einem Softwaresystem gestaltet.
Sicherheitsgewährleistende Anforderungen hingen verifizieren, dass die sicherheitsfunktionalen Anforderungen im laufenden Softwareentwicklungsprozess auch ordnungsgemäß umgesetzt wurden. Dies kann zum Bespiel mittels Softwaretests und entsprechender Dokumentation überprüft werden. [vgl. [15]]
Softwaresicherheitsanforderungen werden in der Literatur auch oftmals als nicht funktionale Anforderungen, Bedingungen (Constraints) oder Sicherheitsrichtlinien bezeichnet und können auf verschiedene Weise erhoben werden. Ein möglicher Ansatz wäre [vgl. [21]]:

1. Modellieren

   Hierbei werden mit einem Team von Softwareentwicklern und Sicherheitsexperten Modelle von zu schützenden Gütern erarbeitet, wie beispielsweise Daten oder Softwarekomponenten. Weiters werden auch mögliche Sicherheitsbedrohungen aufgezeigt und bewertet, welche die Sicherheit der zu schützenden Güter beeinträchtigen könnten.

2. Selektieren

   Danach werden aus dem Softwaresicherheitsmodell benötigte Sicherheitsanforderungen mit den übrigen Anforderungen abgestimmt.

3. Spezifizieren

   Abgeleitete Sicherheitsanforderungen werden in diesem Schritt formal oder informal in einem Dokument spezifiziert.

4. Verifizieren

   Eine anschließende Verifikation der Sicherheitsanforderungen mittels Peer-Review und anderen Validierungsmechanismen prüft diese auf Vollständigkeit, Konsistenz und Korrektheit.

Die Erstellung von sicherer Software ist mehr, als die bloße Definition und Umsetzung von Sicherheitsanforderungen. Es genügt oft nicht, dass beispielweise nur generelle Überlegungen in Hinblick auf Kryptographie oder Passwörter gemacht werden, sondern die richtige Wahl der zu implementierenden Sicherheitsanforderungen ist ebenso immanent. Dabei müssen Sicherheitsanforderungen priorisiert und ausgewählt werden; eine mögliche Art dieser Priorisierung mitsamt Selektion bietet die S^3P-Methode. Der Ansatz basiert auf der Zuweisung von Kosten zu Sicherheitsanforderungen, danach werden die besten Sicherheitsanforderungen (in Berücksichtigung auf die Zielsetzung) mittels Heuristiken ausgewählt. [vgl. [11]]
Sichere Software bedeutet daher, dass die ganze konzipierte Software sicher ist und nicht nur einzelne Teile der Software, welche Sicherheitsanforderungen implementieren. Oftmals wird aber der Umstand vergessen, dass auch Nicht-Sicherheitsanforderungen korrekt funktionieren müssen, um einen reibungslosen Ablauf des Systems zu gewährleisten und es vor möglichen Angriffen zu schützen. [vgl. [14]]
Zur sicheren Entwicklung von Software reicht defensive Programmierung oftmals nicht; Sicherheit muss in die gesamte Entwicklung eingebunden werden. [vgl. [54, 51]]
Speziell bei großen Unternehmen ist die Initiative von sicherer Softwareentwicklung mitunter nicht immer einfach, da viele Überlegungen angestellt werden müssen, wie zum Beispiel in Bezug auf die Ernennung eines Sicherheitsteams, die Wartung von Legacycode, die Planung von Sicherheitsschulungen, die Erstellung von Sicherheitsstandards und Metriken zur Überprüfung der Sicherheit, die Bereitstellung von Tools, die Budgetierung von Softwaresicherheit, das Aufzeigen von möglichen Sicherheitsbedrohungen und so weiter. [vgl. [13]] Zur Erreichung der geforderten Softwaresicherheit ist es daher erforderlich schon im Rahmen der Softwareplanung und -analyse vermehrt Augenmerk auf mögliche Sicherheitsbedrohungen zu legen. Ein möglicher Ansatz wäre zum Beispiel ein bereits bestehendes Klassendiagramm eines zu konzipierenden Systems nach Sicherheitsmustern beziehungsweise Sicherheitslücken zu durchsuchen und eventuelle Schwächen aufzuzeigen. [vgl. [34]]

Nach einer Analyse und Erhebung von Sicherheitsbedrohungen existieren prinzipiell drei Varianten zur Vermeidung von potenziellen Sicherheitsrisiken [vgl. [77]]:

- Verringerung des Risikos

  Das Risiko der erkannten Bedrohung ist sehr hoch und muss verringert werden; es besteht direkter Handlungsbedarf.

- Akzeptanz des Risikos

  Die Bedrohung wurde erkannt und akzeptiert; das Risiko kann indirekt umgangen werden.

- Absicherung gegen das Risiko

  Die Risiken durch Sicherheitsbedrohungen werden auf Dritte umgewälzt.

Nach McGraw [vgl. [54, 76]] können sieben Berührungspunkte zwischen dem Softwareentwicklungsprozess und dem Bereich Softwaresicherheit definiert werden. Die Integration von Softwaresicherheit kann daher in die Punkte [vgl. [54]]

- Sicherheitsanforderungen und Missbrauchfälle
- Design und Risikoanalyse
- Planung von Sicherheitstests
- Implementierung und Codereview
- Durchführung von Penetrationtests
- Feedback und Inbetriebnahme

eingeteilt werden. Diese Punkte werden in der Abbildung 2.1 veranschaulicht, im nachfolgenden Absatz sowie in den darauffolgenden Abschnitten näher erläutert.

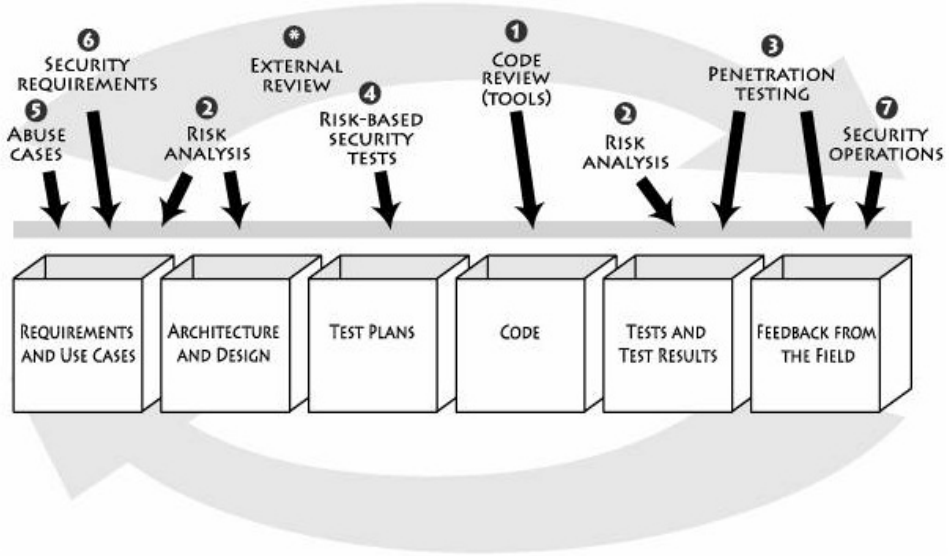

Abbildung 2.1: Berührungspunkte zwischen Softwareentwicklung und Softwaresicherheit [vgl. [54]]

Bei der Anforderungserhebung können Sicherheitsanforderungen miterhoben und basierend darauf Szenarien über den Missbrauch dieser definiert werden. In die Design- und Architekturüberlegungen eines Softwaresystems kann eine Risikoanalyse und ein externes Review miteinbezogen werden. Beim Erstellen von Testfällen für die Software werden auch Sicherheitstests konzipiert, welche nach der Implementierungsphase und entsprechenden Code-Reviews mittels Penetrationtests realsiert werden. Schlussendlich wird ein Testfeedback ausgewertet und geprüft, ob die Software in Betrieb gehen kann. [vgl. [54]]

2.3 Sicherheitsanforderungen und Missbrauchfälle

Analog zur Definition von funktionale und nicht funktionalen Anforderungen ist es ebenso nötig ”Anti-Anforderungen” [McGraw, 2006, [54]] für eine Software zu bestimmen. Diese geben

im Unterschied zu den anderen Anforderungen an, wie sich ein System (z. B. im Fehlerfalle) NICHT verhalten soll. [vgl. [54]] Dabei kann so vorgegangen werden, dass Softwareentwickler einen bestimmten Anwendungsfall konzipieren. Danach wird eine mutwillig durch einen Benutzer verursachte Fehlereingabe in Bezug auf den Anwendungsfall beschrieben und zusätzlich wie sich das Softwaresystem nach der fehlerhaften Eingabe verhalten soll. [vgl. [76]]

2.4 Design und Risikoanalyse

Im Rahmen der Risikoanalyse ist die Erfordernis einer ”Forest-Level” View [McGraw, 2006, [54]] gegeben, wodurch den Sicherheitsanalysten ein Überblick über das System zur Verfügung gestellt wird. Die ”Forest-Level” View kann durch ein UML-Diagramm beschrieben werden und gibt dabei Aufschluss über die Interaktion der kritischen Komponenten eines Softwaresystems. Auf Grund dieser View können Schwachstellenanalysen im Design durchgeführt werden. [vgl. [54]] Ebenso sind die Risiken des Unternehmens zu analysieren in Hinblick auf direkte (z. B. Ausfall des Softwaresystems im Produktivbetrieb) und indirekte (z. B. Ruf- oder Imageschaden) Kosten, die duch Sicherheitsbedrohungen entstehen könnten. [vgl. [76]]

2.5 Planung von Sicherheitstests

Genauso wie funktionale Anforderungen gestestet werden müssen, gilt es auch Sicherheitsanforderungen zu testen. Dabei kann in funktionales und risikobasiertes Sicherheitstesten unterschieden werden. [vgl. [76]] Bei der ersten Art werden die Sicherheitsanforderungen auf ihre Funktionalität überprüft (z. B. ein Verschlüsselungsmechanismus mittels PKI oder die Authentifizierung von Benutzern). Risikobasierte Tests hingegen testen nicht nur, ob Sicherheitsanforderungen funktional korrekt umgesetzt wurden, sondern zielen auf das Testen von Softwarearchitekturschwächen ab. Dabei werden für Sicherheitsanforderungen und ”Anti-Anforderungen” [McGraw, 2006, [54]] mögliche Angriffszenarien ausgearbeitet, welche potenzielle Attacken von Angreifern darstellen könnten. Solch ein Vorgehen bedarf genauer Dokumentation, damit dieses Wissen bei der Durchführung von Penetrationtests angewendet werden kann. [vgl. [54]]

2.6 Implementierung und Codereview

Im Rahmen der Risikoanalyse einer Software können mitunter Schwachstellen im Design festgestellt werden, jedoch wird hier noch keine Rücksicht auf Implementierungsschwächen genommen. Vor allem unerfahrene Softwareentwickler übersehen häufig potenzielle Sicherheitslücken und integrieren dadurch während der Implementierungsphase Sicherheitsfehler in die Software, welche möglicherweise zu Sicherheitsproblemen führen können, wie zum Beispiel Pufferüberläufe oder SQL-Injection. Um dieser Problematik beizukommen, ist es sinnvoll statische Codeanalysen durchzuführen. Mit Hilfe von Tools kann der entstehende Programmcode noch vor der Laufzeit auf Schwächen überprüft werden. Das Problem bei Tools zur statischen Codeanalyse ist, dass sie falsch positive beziehungsweise falsch negative Ergebnisse liefern. Nichts desto trotz sollte statische Codeanalyse Teil einer modernen Softwareentwicklung sein. [vgl. [76, 54]]

2.7 Durchführung von Penetrationtests

Funktionale Sicherheitstests sollen die einwandfreie Funktion von sicherer Software gewährleisten. Es ist aber auch wichtig nicht nur die Software an sich auf Sicherheit zu überprüfen, sondern auch die Konfiguration dieser und die Umgebung in welcher diese installiert und betrieben wird. Penetrationtests erfüllen genau diese Aufgabe und dabei sollen die Erkenntnisse aus der zuvor evaluierten Risikoanalyse zur Anwendung kommen. Penetrationtests können automatisiert

mit Tools (z. B. Fuzzer) durchgeführt werden, welche schädliche Eingaben an die zu testende Software schicken und versuchen diese beispielsweise zum Absturz zu bringen. Tools für Penetrationtests bringen Vorteile: erstens unterstützen sie manuelle Reviews, indem sie Arbeit und Zeit sparen; zweitens können mit Tools Metriken berechnet werden, womit zum Beispiel der Fortschritt der Realisierung und Verbesserung von Softwaresicherheit gemessen werden kann. [vgl. [76, 54]]

2.8 Feedback und Inbetriebnahme

Der letzte Schritt im Entwicklungsprozess von sicherer Software wird durch die Inbetriebnahme beschrieben. Es sollte drauf geachtet werden, dass die Konfiguration und Einsatzumgebung von Software möglichst durchdacht ist; dies wirkt sich positiv auf die Softwaresicherheit aus. Der sichere Betrieb von Software kann beispielsweise realisiert werden mittels Monitoring von Systemereignissen oder durch die adäquate Vergabe von Zugriffsberechtigungen an Benutzer. [vgl. [76, 54]]

Die Entwicklung von sicherer Software erstreckt sich somit über ihren ganzen Entstehungsprozess. Sicherheit kann nicht einfach am Ende des Softwareentwicklungsprozesses hinzugefügt werden, sondern sollte von Anfang an in sämtliche Überlegungen miteinbezogen werden. [vgl. [54]] Es besteht zum Beispiel heutzutage die Möglichkeit mittels agilen Prozessen kontinuierlich Sicherheit in Software zu integrieren. [vlg. [6, 58]]

3 Softwaretesten

Dieses Kapitel behandelt einführend Grundbegriffe aus dem Bereich Softwaretesten. Dabei wird erläutert, wie sich generell der Testprozess von Software gestaltet, welche Arten von Fehler Software beinhalten kann, aber auch nach welchen Aspekten Software speziell getestet werden kann.

3.1 Aspekte beim Softwaretesten

Software kann nach verschiedenen Gesichtspunkten getestet werden, zum Beispiel mit Funktions-, Interface-, Smoke-, Performance- oder Sicherheitstests. [vgl. [24]] Im Rahmen dieser Arbeit werden Sicherheitstests gesondert im Abschnitt 3.3 behandelt.

3.1.1 Funktionstest

Funktionstests testen, ob eine Software ihre funktionalen Anforderungen erfüllt und können prinzipiell als Black-Box- oder White-Box-Tests realisiert werden [vgl. [38, 24]]

Black-Box-Test

Black-Box-Tests gehen davon aus, dass der Tester keine Einsicht in den Quellcode der zu testenden Software besitzt. Er befindet sich vor einer schwarzen Box, die er nach den Anforderungen aus der zugehörigen Softwarespezifikation testet. Somit wird das korrekte Verhalten der Software beispielsweise in Bezug auf Benutzereingaben überprüft. Ein typischer Black-Box-Test ist beispielsweise ein Systemtest. [vgl. [24, 30]] Ein Black-Box-Test wird schematisch in Abbildung 3.1 dargestellt.

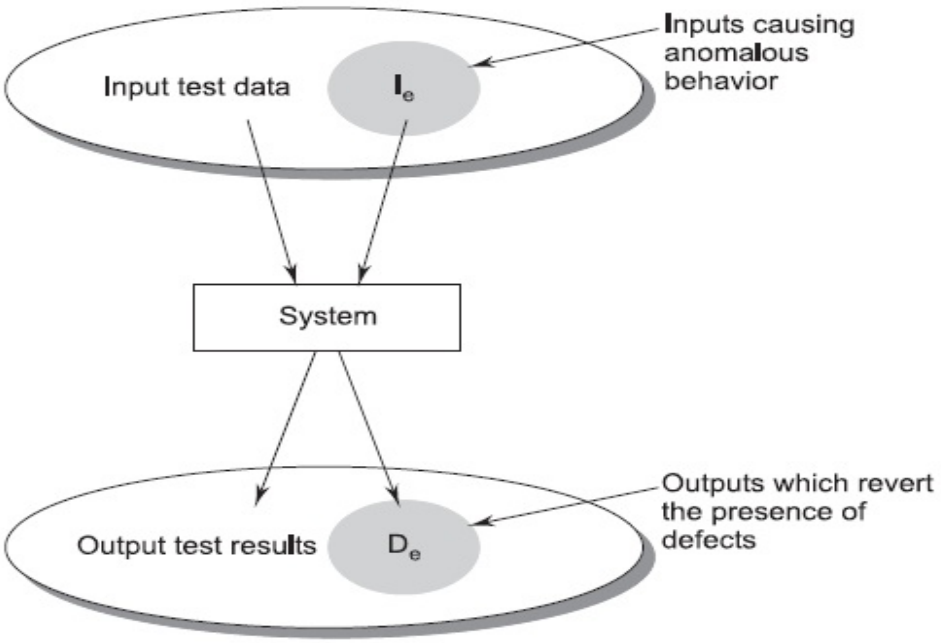

Abbildung 3.1: Schematische Darstellung eines Black-Box-Tests [vgl. [1]]

Black-Box-Tests können zum Beispiel mittels Äquivalenzpartitionierung, Grenzwertanalyse, Entscheidungstabellen oder Transitionsgraphen durchgeführt werden.
Bei der Verwendung von Äquivalenzpartitionierung werden Eingabewerte so zusammengefasst und in äquivalente Klassen eingeteilt, dass die Inhalte dieser Klassen bei einem Softwaretest pro Klasse jeweils dieselbe Ausgabe produzieren.
Der Ansatz der Grenzwertanalyse beschreibt ein anderes Konzept; hierbei wird davon ausgegangen, dass vermehrt Grenzen zwischen verschiedenen Eingabebereichen getestet werden, da in diesen Bereichen häufiger Fehler auftreten.
Mittels Entscheidungstabellen können beispielsweise pro zu testender Softwarekomponente Kombinationen von gültigen und ungültigen Eingabewerten erzeugt werden. Pro Kombination gibt es ein erwartetes Ergebnis, dessen Richtigkeit anschließend im Zuge von Tests überprüft wird.
Testen mit Transitionsgraphen geschieht, indem die zu testende Software als Automat mit verschiedenen Zuständen modelliert wird. Mittels Ereignissen und Daten kann von einem Zustand in den nächsten gewechselt werden; somit können die verschiedenen Softwarefunktionen in Form von erlaubten und nicht erlaubten Transitionen beziehungsweise Zuständen kontrolliert werden. [vgl. [24, 38]]

White-Box-Test

White-Box-Tests - auch strukturelles Testen genannt - werden vom Entwickler der Software selbst ausgeführt und eignen sich daher gut für In-House-Software, weil die Entwickler den Quellcode der zu testenden Software besitzen. Strukturelles Testen wird es deswegen genannt, weil der Tester die Struktur der zu testenden Software vor sich hat. White-Box-Tests werden nicht vorrangig nach der vorliegenden Softwarespezifikation durchgeführt, sondern auf Grund der strukturellen Beschaffenheit des Quellcodes. [vgl. [24]] Das wird schematisch in Abbildung 3.2 zur Schau gestellt.

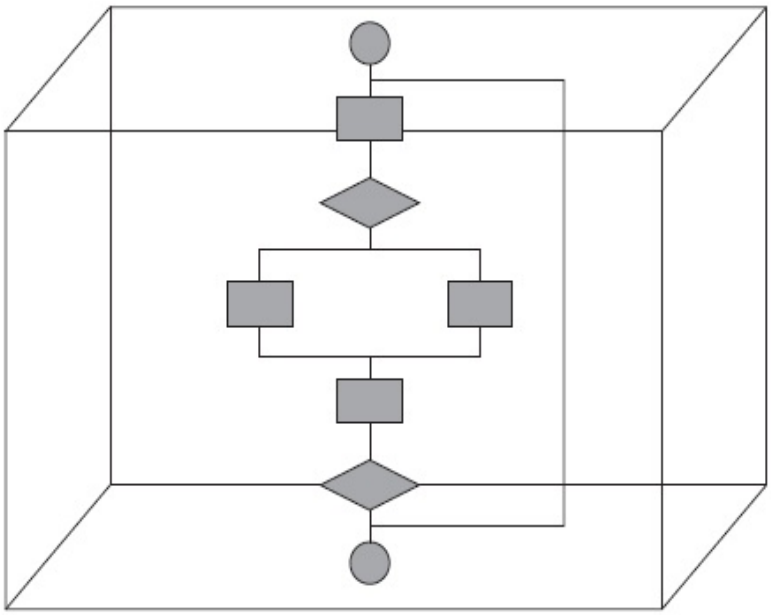

Abbildung 3.2: Schematische Darstellung eines White-Box-Tests [vgl. [1]]

Bei einem White-Box-Test wird die Korrektheit des Datenflusses sowie die Richtigkeit der Ausführungspfade, Bedingungen und Schleifen überprüft. [vgl. [24]] Auf Grund von White-Box-Tests kann zum Beispiel der Entwickler verifizieren, ob alle logischen Bedingungen innerhalb des zu testenden Softwaremoduls mindestens einmal mit wahr und falsch ausgewertet worden sind. Es können auch Datenstrukturen auf ihre Gültigkeit oder Schleifenbedingungen auf Grenzwerte geprüft werden. White-Box-Tests versuchen den entwickelten Quellcode überhaupt lauffähig zu machen. [vgl. [24, 1]]

3.1.2 Performancetest

Im Zuge von Performancetests werden auch die Begriffe Last- oder Stresstests genannt. Diese testen die Software auf nicht funktionaler Ebene, indem sie beispielsweise bei einem Systemtest eine hohe Datenlast erzeugen. Es wird festgestellt wie sich das zu testenden System unter diesen Umständen verhält.
Bei einem Lasttest werden zum Beispiel viele Transaktionen geöffnet oder bei einem Stresstest wird überprüft, ob und wie das System über definierte Grenzen hinaus belastbar ist.
Performancetests messen daher die Antwortwortzeiten und den Durchsatz von Software, wenn diese gerade mit der Abarbeitung von Benutzereingaben oder -anfragen beschäftigt ist. Die Messkriterien für Antwortzeiten werden vom laufenden Geschäftsbetrieb vorgegeben.
Üblicherweise werden Performancetests erst gemacht, wenn der Softwareimplementierungsprozess funktional abgeschlossen ist. Sie werden häufig in den Nachstunden durchgeführt, um den Softwarebetrieb oder die -entwicklung untertags nicht zu behindern. [vgl. [38, 24, 30]]

3.1.3 Interfacetest

Beim Testen von Interfaces wird der Gesichtspunkt auf den Datenaustausch zwischen verschiedenen Quellen gelegt. Das können zum Beispiel Daten sein, welche über das Netzwerk zur zu testenden Software gelangen oder Suchergebnisse aus Datenbankabfragen, die der Benutzer angefordert hat. Ein möglicher Testfall könnte so aussehen, dass die zu testende Software von einer anderen Software Daten erhalten soll. Falls das im Zuge eines Testlaufs nicht der Fall ist, liegt

möglicherweise ein Fehler in der Schnittstellendefinition zwischen dem Sender und Empfänger vor. [vgl. [24]]

3.1.4 Smoketest

Smoketests werden normalerweise im Rahmen der Installation eines Softwaresystems durchgeführt, um zu überprüfen, ob es bei der Installation zu Fehlern (Smoke) gekommen ist. Diese Vorgehensweise kann als integrierender Ansatz betrachtet werden, denn jedesmal, wenn sich in der Software Komponenten ändern oder gar neu hinzu kommen, kann ein Smoketest gestartet werden. Falls dieser Test fehlschlägt, können die Entwickler rasch darüber informiert werden und die aufgetretenen Fehler beheben. Diese Methode ist auch praktisch für Tester, da an diese nur brauchbare, lauffähige Softwareversionen ausgeliefert werden, welche anschließend von diesen getestet werden kann. [vgl. [24, 1, 25]]

3.2 Softwaretestprozess

Software kann allgemein in zwei Bereiche aufgeteilt werden, nämlich: Systemsoftware und Anwendungssoftware. Systemsoftware ermöglicht die Grundfunktionalität von Computern aller Art, allen voran das Betriebsystem und seine diversen Hilfsfunktionen. Anwendungssoftware hingegen setzt auf dem Betriebsystem auf und interagiert direkt mit dem Benutzer (z. B. Computerspiele, Datenbanken, Schreibprogramme usw.) [vgl. [1]]
Anwender von Software berücksichtigen aber oftmals nicht, dass Software Fehler beinhaltet, welche einen reibungslosen Ablauf behindern. Ein Fehlverhalten von Software kann mitunter sogar gravierende Folgen haben (z. B. Absturz eines Flugzeuges). Um solchen Folgen vorzubeugen, müssen Softwaretests durchgeführt werden, welche Software auf ihre Funktionstüchtigkeit überprüfen.

”Testen ist ein Prozess der Programmausführung mit der Intention Fehler zu finden.” [Agarwal et al., 2011, [1]]

Softwaretests können systematisch in Form eines Prozesses durchgeführt werden. Es gibt verschiedene Ansätze zur Testabwicklung, wie beispielsweise jene nach dem Wasserfall- oder Spiralmodell. Im Rahmen dieser Arbeit wird das sich vom Wasserfallmodell ableitende V-Modell näher erläutert. [vgl. [46]]

3.2.1 V-Modell

Beim Ansatz des Wasserfallmodells wird davon ausgegangen, dass die Entwicklung von Software in separaten, abgeschlossenen Phasen erfolgt. Laut Modell können einzelne Phasen nach deren Beendigung nicht mehr erneut durchlaufen werden. Im Zuge der Entwicklung werden zuerst die Anforderungen für die geplante Software erhoben, danach wird das logische, physische und komponentenorientierte Design entworfen. Anschließend erfolgt die Implementierungs- und Testphase der Software. Diese beginnt relativ spät im Softwarelebenszyklus. So können laut Wasserfallmodell beispielsweise zu spät entdeckte, logische Designschwächen in der Software theoretisch nicht mehr rückgängig gemacht werden.
In Anlehnung an das Wassfallmodell wurde daher das V-Modell konzipiert, welches kontinuierliches Testen vorschlägt, anstatt der üblichen Testphase am Ende der Implementierungsphase. Es wird daher so früh wie möglich mit Softwaretests begonnen. [vgl. [46]]
Beim V-Modell wird ausgehend von der Anforderungserhebungsphase ein Testplan erstellt, welcher in jeder darauffolgenden Entwicklungsphase verfeinert wird, da immer mehr Wissen über das zu konzipierende Softwaresystem entsteht. Im Testplan wird festgehalten wie ein bestimmtes Softwaresystem getestet wird. Er beinhaltet verschiedene Testfälle und bei deren Erstellung

können beispielsweise gültige beziehungsweise ungültige Werte oder Grenzwerte als Testeingaben verwendet werden. Eine andere Möglichkeit Testfälle zu generieren, besteht darin, das Benutzer(fehl)verhalten zu analysieren und in Form von Tests niederzuschreiben. [vgl. [2]] Des Weiteren ergeben sich Testfälle auch aus den fachlich dokumentierten Softwareanforderungen. Nach der Erfassung von Testfällen werden Testskripts erzeugt, welche zum Start von Softwaretests verwendet werden. [vgl. [46]] Eine schematische Ansicht des V-Modells gibt die Abbildung 3.3 wider.

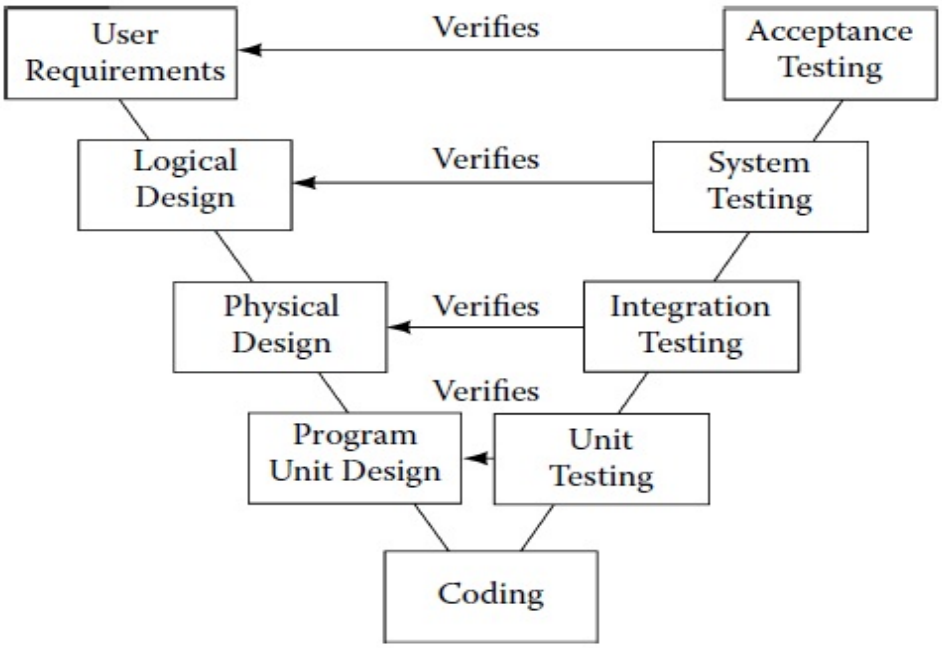

Abbildung 3.3: Softwaretestprozess mittels V-Modell [vgl. [46]]

Das V-Modell beeinhaltet verschiedene Stufen, um Software entsprechend zu testen. Diese heißen:

- Modultest (Unit-Test)
- Integrationstest
- Systemtest (Akzeptanztest)

Jeder Stufe des Softwareentwicklungsprozesses wird eine Teststufe gegenüber gestellt. Somit kann schon im Vorfeld überprüft werden, welche Fehler sich zum Beispiel in den kleinsten Softwareeinheiten befinden, bevor diese zu größeren zusammengefasst und miteinander getestet werden. Diese einzelnen Teststufen werden im Abschnitt 3.2.3 näher erläutert.

3.2.2 Softwarefehler

Softwaresicherheit wird durch Fehler verschiedener Art innerhalb von Software beeinträchtigt. Dieses Fehlverhalten kann von potenziellen Angreifern benützt werden, um Schaden anzurichten und die Sicherheit von Daten zu gefährden. Softwarefehler können durch verschiedene Gegebenheiten in die Software implementiert werden. Einerseits kann mangelndes Know-How bei Softwareentwicklern zu einer Implementierung von Sicherheitslücken führen.

Dabei handelt es sich zum Beispiel um [vgl. [43]]:

- eine fehlende Validierung von Benutzereingaben,
- Fehler bei der Authentifizierung von Benutzern und Vergabe von Berechtigungen an Benutzer,
- Fehlerhaftigkeit im Mehrbenutzerbetrieb (z. B. zwei Benutzer schreiben zeitgleich auf ein und denselben Datensatz),
- mangelnde Fehlerbehandlung (z. B. Dateien werden vor dem Öffnen nicht auf ihre Existenz geprüft) oder
- schlechtes Softwaredesign.

Anderseits können unerfahrene Softwaretester diese Sicherheitslücken durch suboptimale Testfälle nicht auffinden. In weiterer Folge bleiben Softwarefehler unentdeckt. [vgl. [43]] Zum Begriff Softwarefehler lassen sich in der englischsprachigen Literatur mehrere Ausdrücke finden, nämlich: ”Bug”, ”Flaw”, ”Defect”. [vgl. [54]]
Ein Defekt (Defect) definiert generell den Begriff Fehler, welcher sich innerhalb der Software auf der Design- oder Implementierungsebene befinden kann.
Ein Fehler (Bug) beschreibt im Fachjargon einen Fehler auf der Implementierungsebene. Dabei kann es sich beispielsweise um fehlerhaften Softwarecode handeln, der bis zum Auftreten des Fehlers noch nie ausgeführt wurde und somit unentdeckt geblieben ist.
Mängel (Flaw) hingegen stellen Fehler auf der Designebene dar, welche durch unzureichendes Softwaredesign verursacht werden. Dies können zum Beispiel Designschwächen bei der Fehlerbehandlung (z. B. das Öffnen einer Datei schlägt fehl) oder die unsichere Konzeption von Logging-Mechanismen sein. [vgl. [54]]
Fehler können aber nicht nur in der Softwareentwicklung auftreten, eine andere Art von Fehlern sind Eingabefehler. Sie beziehen sich nicht primär auf das fehlerhafte Design oder die falsche Implementierung von Software, sondern auf menschliches Fehlverhalten durch den Benutzer. Falls Eingabefehler auftreten, kann dies zu einem (vom spezifizierten) abweichenden Softwareverhalten führen. Fehlerhafte Eingaben führen mitunter zur Aufdeckung sowie Auslösung von Defekten, Fehlern und Mängeln, welche in weiterer Folge die Sicherheit von Software und Daten gefährden können. [vgl. [25]] Wie bereits erwähnt, kann Testen die vollständige Fehlerfreiheit von Software nicht gewährleisten. Es ist jedoch möglich mit Hilfe der nachfolgenden Teststufen Defekte, Fehler und Mängel in Software zu finden.

3.2.3 Softwareteststufen

In diesem Abschnitt werden die Teststufen von Software näher erklärt, nämlich Modultest, Integrationstest und Systemtest.

Modultest (Unit-Test)

Das Testen von Einheiten oder so genannten Units beginnt auf der ersten Stufe. Hierbei werden die kleinsten Einheiten einer Software exklusiv und unabhängig voneinander getestet, es muss daher keinerlei Information über die Kapselung der Software bekannt sein, noch wie die Komponenten zusammenspielen. [vgl. [4]] Ein Modultest wird schematisch in der Abbildung 3.4 dargestellt.

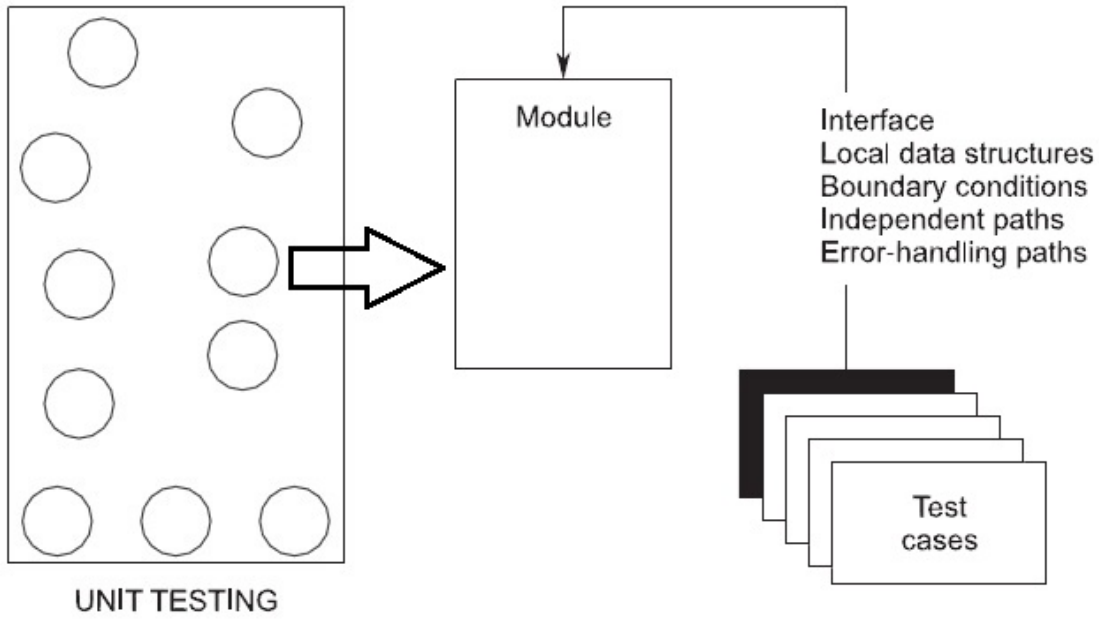

Abbildung 3.4: Schematische Darstellung eines Modultests [vgl. [1]]

Bei Unit-Tests wird eine Softwareeinheit per Schnittstelle getestet, um das Verhalten der getesteten Komponente beispielsweise im Fehlerfall zu überprüfen. Zum Beispiel kann auch verifiziert werden, ob es innerhalb des zu testenden Codemoduls Endlosschleifen gibt oder ob jeder Ausführungspfad zumindest einmal durchlaufen wurde. [vgl. [1]]
Es gibt verschiedene Ansätze, um Unit-Tests zu realisieren. Einen davon stellt das Test-Driven-Development dar, wo vor der eigentlichen Entwicklung der Komponente bereits Tests erstellt werden, um die Funktion dieser Einheit danach zu testen. Dadurch wird verhindert, dass sich frühzeitig Fehler in die Entwicklungsphase der Software einschleichen. [vgl. [2, 74]]

Integrationstest

Beim Integrationstest werden mehrere Softwarekomponenten zusammengeschlossen und getestet. Dabei wird zum Beispiel die Interaktion von Softwarekomponenten mit anderer Software, wie beispielsweise dem Datei- oder Betriebsystem getestet. Es wird davon ausgegangen, dass die einzelnen Softwarekomponenten für sich bereits funktionieren. Eine schematische Darstellung eines Integrationstests zeigt die Abbildung 3.5.

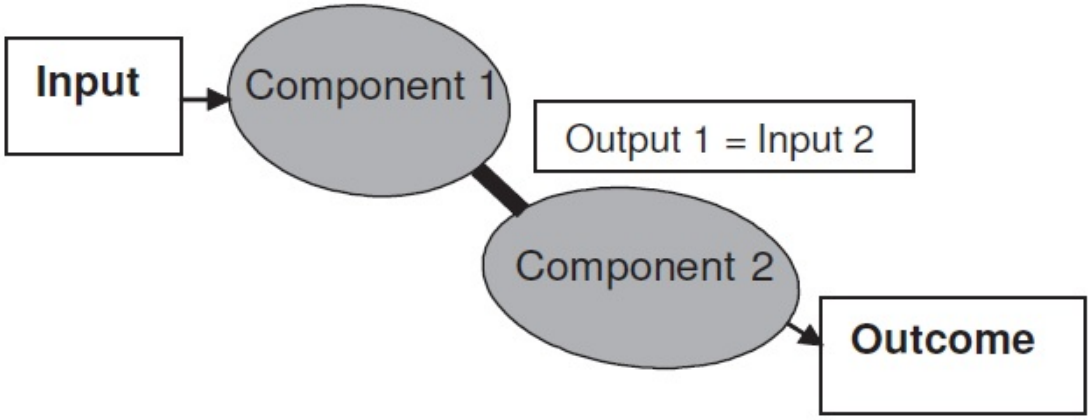

Abbildung 3.5: Schematische Darstellung eines Integrationstests [vgl. [38]]

Integrationstests können in Komponenten- und Systemintegrationstests unterschieden werden. Bei der Komponentenintegration werden die Schnittstellen zwischen zwei oder mehr Komponenten derselben Software miteinander getestet. Im Rahmen der Systemintegration wird die Interaktion eines ganzen Softwaresystems mit anderen Systemen getestet. [vgl. [38, 30]]

Die Integration von Software kann nach folgenden Vorgehensweisen erfolgen [vgl. [30]]:

- top-down
- bottom-up
- funktional-inkrementell
- Big-Bang

Beim top-down Ansatz wird davon ausgegangen, dass jeweils die Komponenten beginnend auf oberster Ebene getestet werden, alle sich darunter befindlichen Komponenten werden von Prototypen simuliert beziehungsweise ersetzt.
Das Vorgehen nach bottom-up gestaltet sich analog dazu, nur in die andere Richtung. Hier werden alle Komponenten beginnend auf der untersten Ebene getestet und die sich darüber befindlichen imitiert.
Die funktional-inkrementelle Intregration definiert das Vorgehen so, dass die Software nach ihren Funktionen integriert wird, unabhängig davon, ob sich diese mit der statischen Struktur des Systems decken.
Beim Big-Bang werden alle Komponenten auf einmal integriert. Das hat den Vorteil, dass vor der Integration alle Komponenten bereits fertig sind. Der Nachteil dieser Methode ist, dass auf Grund der zu späten Integration aller Komponenten sehr viele Fehler auf einmal auftauchen, welche mitunter sehr hohe Reparaturkosten verursachen können. [vgl. [61, 30]]

Systemtest (Akzeptanztest)

Ein Systemtest stellt den Test des gesamten, implementierten Softwaresystems dar. Dabei werden nicht, wie fälschlicherweise oft angenommen, einzelne Funktionen des Systems getestet sondern, ob das ganze System den spezifizierten Anforderungen im Anforderungsdokument entspricht. [vgl. [55]] Ein System- beziehungsweise Akzeptanztest wird schematisch durch Abbildung 3.6 veranschaulicht. [vgl. [38]]

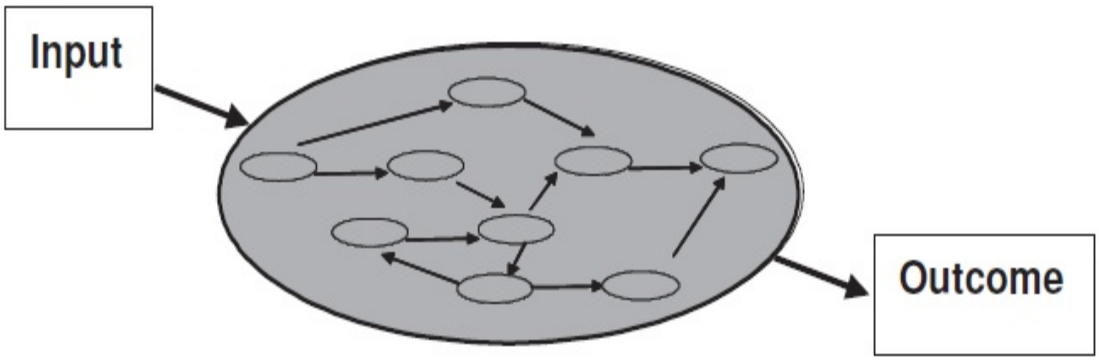

Abbildung 3.6: Schematische Darstellung eines Systemtests [vgl. [38]]

Systemtests können in drei verschiedene Kategorien eingeteilt werden [vgl. [1]]:

- Alphatest
- Betatest
- Akzeptanztest

Alphatests finden prinzipiell auf der Seite der Softwareentwicklung statt. Eine Entwicklungsumgebung mitsamt Installation der zu testenden Software wird ein paar Testern zur Verfügung gestellt und diese können anschließend das Softwareprodukt testen. Diese Variante von Systemtest ist in Bezug auf das Finden von Fehlern relativ eingeschränkt, da sich die Software in einer simulierten Umgebung und nicht im Echtbetrieb befindet. Die geringe Anzahl an Testern trägt ebenso dazu bei, dass sich das Finden von Fehlern in Grenzen hält. Beim Alphatest muss das Entwicklungsteam nicht permanent anwesend sein. Außerdem werden erhebliche Kosten verursacht, um die simulierte Echtzeitumgebung für das zu testende System bereitzustellen.
Beim Betatest von Software wird diese an einen bestimmten Benutzerkreis ausgeliefert. Eine Installation für den Echtbetrieb ist jedoch auch hier nicht vorgesehen. Die Benutzer testen die Software und melden so gefundene Fehler dem Entwicklungsteam. Diese Fehler gelangen anschließend in einen so genannten Change-Management-Prozess, in welchem sie abgearbeitet werden. Als Resultat wird eine neue, verbesserte Version der Software herausgegeben. Betatests dauern oft über einen längern Zeitraum, der Benutzer erhält sozusagen ziemlich früh Einblick in das entwickelte System, welches aber noch nicht ganz ausgereift ist.
Eine andere Form von Systemtests stellen Akzeptanztests dar, welche letztendlich bestimmen, ob der Endbenutzer die entwickelte Software verwendet oder ihre Verwendung verweigert. Der Endbenutzer soll daher im Rahmen dieser Tests mit der Software vertraut werden. In dieser Teststufe wird ebenso verifiziert, ob das Softwaresystem bereit für den Echteinsatz ist. [vgl. [1, 30]]

3.3 Verifikation von Softwaresicherheit

Es gibt verschiedene Arten, um die Sicherheit von Software zu verifizieren. Allen voran sind solche Verfahren Penetrationtests [vgl. [54]], welche beispielsweise mit Fuzzern durchgeführt werden. Generell können sechs Arten von Penetrationtests unterschieden werden [vgl. [25, 39]], nämlich: Black-Box, Double-Black-Box, Gray-Box, Double-Gray-Box, Tandem-White-Box und Reversal. Im nächsten Abschnitt werden Ansätze zum automatisierten Sicherheitstesten mittels Fuzzern näher erläutert.

3.3.1 Testautomatisierung mittels Fuzzing

Softwaresicherheit kann mittels entsprechender Tests verifiziert werden. Zum Beispiel kann heutzutage für solche Zwecke Fuzzing mit Hilfe von entsprechenden Fuzzern eingesetzt werden. Die Definition von Fuzzing ist jedoch nicht immer ganz eindeutig; es stellt eine Art von Negativtests dar, welche versuchen durch fehlerhafte Testdaten Software zum Absturz zu bringen. [vgl. [71]] Fuzzing kann aber auch im Bereich der Grenzwertanalyse angesiedelt werden, da versucht wird Testfälle zu erzeugen, welche inhaltlich zwischen den Grenzen der von erlaubten und nicht erlaubten Eingabewerten eines zu testendes Systems liegen. [vgl. [69]]
Prinzipiell lässt sich Fuzzing in zwei Gebiete einteilen, nämlich: mutationsbasiertes Fuzzing und erzeugungsbasiertes Fuzzing. [vgl. [69]] Beim mutationsbasierten Fuzzing kann Software auf Fehler überprüft werden, indem einzelne Bits der Eingabewerte verändert werden. [vgl. [71]] Beim erzeugungsbasierten Fuzzing werden auf Grund von Protokollen oder Dateiformaten Testfälle erstellt. Eine solche Art von Testfällen ermöglicht es die Software auf verschiedenen Ebenen zu testen (z. B. Datenbankschicht, Präsentationsschicht usw.) [vgl. [69, 71]]
Fuzzing kann über Black-Box- und Gray-Box-Tests realisiert werden; dabei wird so vorgegangen, dass das zu testende System über verschiedene Schnittstellen mit kompromittierenden Eingabewerten versehen wird. Es muss nicht zwingend Information über das Innenleben des zu testenden Softwaresystems bekannt sein. [vgl. [71]]

Aufbau von Fuzzern

Fuzzer besitzen üblicherweise die folgenden, logischen Komponenten, welche verschiedene Funktionen zum Aufspüren von Softwarefehlern beinhalten [vgl. [71]]:

- Protokollmodellierung

  Im Rahmen der Protokollmodellierung werden verschiedene Datenformate für die Schnittstellen der zu testenden Software festgelegt. Diese Formate geben an, wie die Eingabewerte (in Form von Nachrichten) übertragen werden und zur Software gelangen. Für die Definition von Formaten können kontextfreie Grammatiken verwendet werden.

- Anomaliebibliotheken

  Fuzzer können generell mit vordefinierten Werten, so genannten Attack-Vektoren, arbeiten oder mit Hilfe von randomisiert erzeugten Testdaten.

- Angriffssimulation

  Diese Komponente ermöglicht den Angriff von Software über die zuvor bestimmten Attack-Vektoren und Protokolle.

- Laufzeitanalyse

  Bei der Laufzeitanalyse wird das zu testende System und seine Laufzeitumgebung kontrolliert und beobachtet.

- Auswertung

  Das Testergebnis vom Fuzzing-Prozess kann mit der Auswertungskomponente aufbereitet werden und Entwicklern oder Analysten zur Verfügung gestellt werden, um die gefundenen Fehler zu analysieren, zu bewerten und zu beheben.

- Dokumentation

  Sowohl der Fuzzer als auch die generierten und ausgeführten Testfälle sollten zwecks Bedienung und Nachverfolgbarkeit dokumentiert werden.

Fuzzing-Phasen

Beim Fuzzing gibt es keine richtige oder falsche Vorgehensweise, sondern diese hängt allein von der zu testenden Applikation, den Erfahrungen des Testers und den Eingabeformaten ab, welche getestet werden sollen. [vgl. [69, 59]] Fuzzing kann in folgende Phasen unterteilt werden [vgl. [69, 59]]:

- Zielidentifizierung

  Im ersten Schritt gilt es das Ziel zu identifizieren, welches mittels Fuzzing getestet wird. Dabei kann unterschieden werden, ob eine ganze Applikation oder nur Bibliotheken einer Software getestet werden. Bei Bibliotheken von Dritten sollte darauf geachtet werden, ob es eine historische Aufzeichnung von Schwachstellen in deren Software gibt, diese gilt es zu beachten. Falls es keine historischen Informationen dazu gibt, kann davon ausgegangen werden, dass die Bibliothek möglicherweise schlecht implementiert beziehungsweise getestet wurde.

- Identifizierung der Eingabewerte

  Beim Fuzzing ist es wichtig, dass vor dem Testlauf mögliche Eingabewerte für die zu testende Software überlegt werden. Diese Eingabewerte werden auch Eingabevektoren genannt. In den meisten Fällen stammen die Eingabewerte aus Dateien, Registryeinträgen, APIs, Datenbanken oder kommen über eine Netzwerkschnittstelle.

- Datenerzeugung

  Nach den zuvor angestellten Überlegungen müssen Fuzzing-Daten erzeugt werden. Dabei kann auf Ansätze zurückgegriffen werden, wie jenen von vordefinierten, mutierten oder dynamisch generierten Eingabewerten. Dieser Vorgang sollte möglichst automatisiert gestaltet sein, wobei jedoch darauf zu achten ist, dass die Daten eine gewisse Grundgültigkeit besitzen, da andernfalls die Parsing-Schnittstelle der zu testenden Software die Eingabe verwerfen könnte. Die Daten sollten daher syntaktisch richtig, aber semantisch fehlerhaft sein.

- Durchführung

  Dieser Abschnitt geht einher mit dem vorherigen; dabei werden die erzeugten Daten in Form von Paketen an die zu testende Software geschickt. Bei diesem Prozess ist Automatisierung unbedingt notwendig, da Fuzzing andernfalls nicht effizient genug betrieben werden könnte.

- Überwachung

  Die mittels Fuzzing getestete Applikation muss überwacht werden, weil andernfalls nicht ermittelt werden kann, welche Datenpakete zu einem möglichen Fehler beziehungsweise Systemabsturz geführt haben.

- Verwendbarkeit

  In dieser Phase wird festgestellt wie gefundene Fehler weiter zu behandeln sind beziehungsweise, ob es zum Beispiel möglich ist, dass ein konkreter Fehler noch andere Fehler impliziert oder bedingt hat.

Funktionsweise von Fuzzing

Die Vorgehensweise beim Fuzzing besteht darin, Nachrichten zwischen dem Fuzzer und dem zu testenden System (SUT) auszutauschen. Diese Nachrichten sind Testfälle mit entsprechenden Eingabewerten, deren Verarbeitung durch das zu testende System getestet wird. Dabei sendet der Fuzzer Eingabewerte im Zuge eines Testlaufs zur Software hin, welche je nach Testfall bestimmte Ausgabewerte produziert. Diese werden an den Fuzzer zurückgeschickt und in Form von Testergebnissen abgespeichert. Antwortnachrichten vom zu testenden System an den Fuzzer können unterschiedlich sein. Dabei kann es sein, dass die erfolgte Eingabe gültig war und eine gültige Antwort zurück kommt. Es kann aber auch vorkommen, dass die Eingabe zwar syntaktisch (z. B. auf Protokollebene) richtig, aber semantisch falsch war, ein Eingabefehler erkannt und eine entsprechende Fehlermeldung an den Fuzzer zurückgeschickt wurde. In einem weiteren Fall kann der Fuzzer aber auch unerwartete Antwortnachrichten (z. B. falsches Nachrichtenformat) erhalten, welche mit Hilfe von korrupten Eingaben und dadurch ausgelösten Fehlern zustande gekommen sind. Falls der Fuzzer jedoch auf Eingabewerte keine Antwort vom System erhält, kann davon ausgegangen werden, dass ein Systemabsturz erfolgt ist. [vgl. [71]] Ein schematischer Fuzzing-Vorgang wird in der Abbildung 3.7 dargestellt.

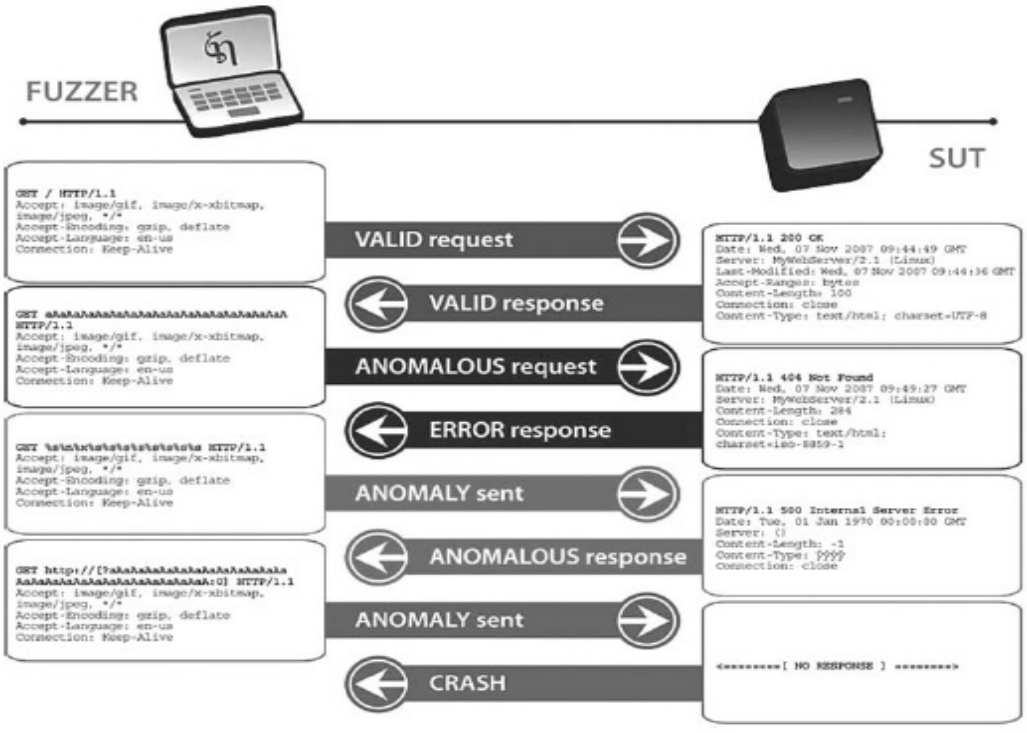

Abbildung 3.7: Schematische Darstellung der Durchführung des Fuzzing-Prozesses [vgl. [71]]

Während des Fuzzing-Vorganges wird das zu testende System ständig überwacht. Es können die erhaltenen Antworten als Testergebnisse für eine spätere Analyse gespeichert werden. Der Fuzzer bewertet nach bestimmten Kriterien, ob die durchgeführten Testfälle Fehler produziert haben. Falls Fehler gefunden wurden, könnten diese ein Indiz für Sicherheitslücken im getes-

teten Softwaresystem sein, welche repariert werden müssen. [vgl. [71]] Fuzzing wird aktuell in verschiedenen Bereichen zum Testen von Software eingesetzt.
Eine Variante von Fuzzing stellt das H-Fuzzing dar; hierbei werden Web-Services getestet. Eine generelle Schwäche von Fuzzing ist die Generierung von zufälligen Eingabewerten. Es besteht daher beim Test der Abfrage "if(y == 2) then" nur eine Chance von 1 zu 2^{32}, dass die 32-bit Variable y den Zahlenwert 2 enthält, und der Pfad ausgeführt wird. H-Fuzzing verbessert die Schwächen eines solchen Fuzzing-Prozesses, indem zum Beispiel vor einem Fuzzing-Testlauf der zu testende Code statisch analysiert wird. Dabei werden sämtliche Abarbeitungspfade des zu testenden Codes untersucht und von dieser Information ausgehend ein Baum aufgebaut. Anschließend werden willkürliche Eingabewerte für das zu testende Programm festgelegt und durch den Abarbeitungsbaum durchgereicht. Somit können zu jedem Pfad im Baum passende Eingabewerte erzeugt werden. Die Wahl des nächsten Pfades ergibt sich nach dem Tiefensuche-algorithmus. Fuzzing erreicht dadurch eine höhere Coverage-Rate. [vgl. [28]]
Gewisse Softwarefehler treten nur bei bestimmten Rahmenbedingungen auf, welche es ebenfalls zu testen gilt. Um solche Situation zu testen, kann Configuration-Fuzzing verwendet werden; hierbei handelt es sich um einen Ansatz bei dem die Konfiguration eines laufenden Softwaresystems randomisiert verändert wird. Es wird in einer anderen Laufzeitumgebung parallel zum laufenden Betrieb einer Software eine neue Instanz dieser Software installiert. Danach wird die Konfiguration variabel verändert und eine zu testende Funktion ausgewählt. Nach dem Test dieser Funktion wird mitprotokolliert, ob es eine Veränderung gegenüber des ursprünglich angeforderten Verhaltens dieser Funktion gibt. [vgl. [16]]
Einen anderen, theoretischen Ansatz zum Fuzzing bietet Smart-Fuzzing. Diese Methode beschreibt die bereits erwähnten Fuzzing-Phasen etwas anders. Im Rahmen der Zielidentifizierung könnte beispielsweise so vorgegangen werden, dass auf Bytecodeebene nach Schwachstellenmustern gesucht wird. Zusätzlich wird noch eine Analyse durchgeführt mit welcher herausgefunden werden soll, wo in der zu testenden Software Daten von nicht vertrauenswürdigen Quellen bezogen werden (z. B. Benutzereingaben oder aus einem Netzwerk stammende Daten). In der Phase Verwendbarkeit wird versucht zu überprüfen, ob Fehler andere bedingen; dabei kann eine Stack- oder Heapanalyse von Vorteil sein. [vgl. [7]]

4 Verarbeitung von Datenmengen mittels Data-Mining

In diesem Kapitel werden Grundbegriffe und Definitionen aus dem Bereich Data-Mining erklärt. Es wird erklärt auf welche Daten Data-Mining angewendet werden kann. Dabei wird die Vorbereitung und Darstellung von Wissen beschrieben. Die Verarbeitung von Wissen, im speziellen Fall aus dem Bereich Softwaresicherheit, wird gesondert im Kapitel 6 behandelt.

4.1 Data-MiningProzess

Die Menge an Datenbeständen wächst weltweit rasant an. Dieser Wachstumsprozess betrifft Daten aller Art. Zum Beispiel kann es sich um Daten aus dem Berufsleben handeln. Es können aber auch persönliche Daten davon betroffen sein. Computer und die globale Vernetzung des Internet ermöglichen die Speicherung all dieser Datenbestände. Es sei hier auf die ständig zunehmenden Festplattenkapazitäten verwiesen. Auf Grund dieser hohen Kapazitäten können beispielsweise diverse Internetserviceanbieter Terabytes an Daten speichern. Eine Vielzahl von Software bietet zum Beispiel eine Unterstützung bei Einkäufen im Internet an. Mit Hilfe solcher Software können Unternehmen unter anderem das Kaufverhalten, die Wohnanschrift oder finanzielle Eigenschaften ihrer Kunden erfassen und speichern.
Nach der Speicherung der Daten befinden sich diese zunächst in einer Art Rohzustand. Diesen gilt es mit unterschiedlichen Methoden zu raffinieren. Aus dem raffinierten Zustand der Daten können nützliche Informationen extrahiert und in weiterer Folge für verschiedene Auswertungen sowie als Entscheidungsgrundlage verwendet werden. Dieser Prozess wird generell als Data-Mining bezeichnet und ist daher wie folgt definiert:

”Data Mining ist eine Sammlung von Techniken zur effizienten und automatisierten Analyse von zuvor unbekannten, gültigen, neuen, nützlichen und nachvollziehbaren Mustern in großen Datenmengen. Die ermittelten Muster müssen nachweislich so aufbereitet werden, dass sie in einem Unternehmensentscheidungsprozess verwendet werden können.” [Gupta, 2006, [32]]

Der Begriff Data-Mining kommt aus dem Bereich der Statistik und wird oft in Verbindung mit maschinellem Lernen gesehen. Es wird davon ausgegangen, dass Daten eine bestimmte statistische Verteilung zu Grunde liegt. Diese kann maschinell mit speziellen Algorithmen analysiert und ausgewertet werden, wie zum Beispiel mit Hilfe von Entscheidungsbäumen oder durch Clusteranalyse. Data-Mining kann prinzipiell auf Daten aller Art angewendet werden und je nach Herkunftsquelle der Daten in die Begriffe Data-Mining, Web-Mining oder Text-Mining eingeteilt werden.
Data-Mining bezeichnet dabei den Basisprozess, um Information zu extrahieren und analysieren. Dieser kann je nach Datenquelle adaptiert werden. Beim Web-Mining bezieht sich die Datenherkunft auf das Internet. Hier kann beispielsweise der Inhalt oder die Linkstruktur von Internetseiten analysiert werden. Text-Mining hingegen konzentriert sich auf die Analysen von unstrukturierten Texten. [vgl. [5]] Je nach Größe des Datenbestandes können verschiedene Data-Mining-Methoden zum Einsatz kommen. Meistens wird beim Data-Mining von großen Datenbeständen ausgegangen, da hierbei die Wahrscheinlichkeit höher ist, dass interessante, wertvolle Information gefunden und weiter verarbeitet werden kann. [vgl. [32, 44, 65, 75]]

Der Data-Mining-Vorgang kann in einzelne Schritte unterteilt werden [vgl. [60, 29, 35]]:

1. Datenzugriff und -bereinigung

   Bevor mit Data-Mining begonnen werden kann, sollten Überlegungen in Bezug auf die durch den Prozess angestrebte Zielerreichung (z. B. Auswertung der Profitabilität von Produkten) gemacht werden, da alle weiteren Schritte darauf aufbauen. Erst danach kann der Bereinigungsprozess der zu analysierenden Daten gestartet werden. In diesem Schritt wird versucht die Daten in einen konsistenten Zustand zu bringen und Fehler im Datenbestand zu korrigieren.

2. Datenintegration

   Daten kommen oftmals aus sehr unterschiedlichen Quellen. Diese befinden sich in diesem Schritt bereits in einem raffinierten Zustand, müssen jedoch noch zusammengeführt werden. Zu diesem Zweck werden Data Warehouses konzipiert, in welche die Daten anschließend integriert werden.

3. Datenselektion und -transformation

   Je nach zuvor definierter Zielfunktion werden alle für eine Analyse relevanten Daten aus dem bereits bereinigten und integrierten Datenbestand im Data-Warehouse ausgewählt. Im Zuge der Transformation werden die selektieren Daten in eine andere Darstellung konvertiert. Möglicherweise gibt es bei der Transformation auch noch Gründe für eine weitere Reduktion der zu Grunde liegenden Daten, welche mitunter den Mining-Prozess vereinfacht. Des Weiteren können diverse, statistische Methoden ausgewählt werden, welche im Nachfolgeschritt zur Anwendung kommen. Es werden dabei deskriptive und prädiktive Methoden unterschieden. Deskriptive Verfahren beschreiben die Daten und kategorisieren sie nach ihren Eigenschaften; prädiktive wiederum versuchen Daten und die Beziehung zwischen einzelnen Elemente innerhalb dieser Daten zu erforschen. Sie können auch für Prognosen beim maschinellen Lernen eingesetzt werden. Die Vorgänge Datenzugriff beziehungsweise -bereinigung, -integration und -transformation werden in der Fachliteratur auch häufig als ETL-Prozess (Extraktion, Transformation and Loading) bezeichnet. [vgl. [17, 31]]

4. Data-Mining und Wissensrepräsentation

   Nachdem die statistischen Methoden ausgewählt und evaluiert wurden, kann die Implementierung von Algorithmen beginnen. Die zuvor ausgewählten Algorithmen werden bei der Verarbeitung auf die zu analysierenden Datenbestände angewendet, um interessantes Wissen aus den Daten zu extrahieren. Dieses Wissen bildet die Grundlage für darauf aufbauende Entscheidungen. Im Rahmen der Wissensdarstellung wird das gewonnene Wissen in eine anschauliche Form gebracht und einem Interessentenkreis zur Verfügung gestellt.

4.2 Daten und ihre Herkunft

Daten können aus vielfältigen Datenquellen stammen. In diesem Zusammenhang seien relationale Datenbanken, Data-Warehouses, Log-Dateien, Datenstreams und das World-Wide-Web erwähnt. [vgl. [32, 57]] In dieser Arbeit wird aber nur auf relationale Datenbanken und Data-Warehouses näher eingegangen.

4.2.1 Relationale Datenbanken

Datenbankmanagementsysteme – kurz DBMS – werden zur Verwaltung von Datenbanken verwendet. DMBS sind für die Verteilung, Wartung, physische Speicherung, Wiederherstellung, Transaktionssteuerung und Konsistenzerhaltung von Daten verantwortlich. Normalerweise bestehen Datenbanken aus mehreren Datendateien. Diese können wiederum Strukturen beinhalten, wie Tabellen, Attribute, Relationen, Indizes und noch viele andere Metadaten. Ein DBMS muss daher auch die Möglichkeit zur Definition und Verwaltung solcher Strukturen bieten. Außerdem muss das DBMS die Berechtigungen für den Datenzugriff steuern, um eine missbräuchliche Verwendung der Daten zu verhindern.
Relationale Datenbanken arbeiten im Prinzip mit Tabellen. Diese können logisch miteinander verknüpft sein und werden in so genannten Tablespaces gespeichert. Tabellen bestehen aus Zeilen und Spalten; dabei beschreiben die Zeilen einzelne Objekte innerhalb der Tabelle. Spalten hingegen spiegeln die Eigenschaften dieser Objekte in Form von Attributwerten wider. Ein Objekt besitzt einen eindeutigen Schlüssel, der es identifiziert. Ein DBMS bietet auch die Möglichkeit Tabellen miteinander zu verknüpfen. Ein weiteres Merkmal von relationalen Datenbanken sind Einschränkungen (Constraints). Damit kann die Integrität der Daten überprüft werden.
Eine Datenbank kann mittels ER-Diagramm (Entity-Relationship-Diagramm) abgebildet werden. Die Entitäten beschreiben Objekte in der realen Welt und deren Beziehungen zueinander. Data-Mining kann auf relationale Datenbanken angewendet werden. Es wird dabei mit Hilfe von Datenbankabfragen vorgegangen. Solche Abfragen können Aggregatfunktionen enthalten, welche die Summe, den Durchschnitt, die Anzahl, den Maximal- oder Minimalwert bestimmter Datensätze ermitteln. [vgl. [57, 17, 35]]

4.2.2 Data-Warehouse

Viele Unternehmen sammeln eine Unzahl von Daten über ihre Kunden. Das Kundenverhalten soll studiert werden, um daraus den größten möglichen Profit zu erzielen. Dazu müssen die Daten zuallererst aus unterschiedlichen Quellen zusammengefasst und anschließend ausgewertet werden. Dieser Vorgang wird Data-Warehouse-Prozess genannt.
Ein Data-Warehouse kann als eine Art Lager oder Archiv betrachtet werden, das seine Dateninhalte von unterschiedlichen Quellen bezieht, wie beispielsweise relationalen Datenbanken. Data-Warehouses werden durch den bereits erwähnten ETL-Prozess erzeugt. [vgl. [68]] Dabei werden die aufbereiteten Daten in einem einheitlichen Schema zur weiteren Verarbeitung und Analyse gespeichert. Data-Warehouses ermöglichen es Unternehmen das Kundenverhalten über eine gewisse Zeit zu erforschen. Die gespeicherten Daten können in einer historischen Sicht von mehreren Jahren betrachtet werden. Zu diesem Zweck werden die Daten nach Kriterien unterteilt und mittels Dimensionen entsprechend aggregiert dargestellt. Eine Dimension könnte zum Beispiel die Summe aller Erträge oder die Absatzmenge von Produkt A des Sortiments über einen festgelegten Zeitraum sein. Diese Art der Datenansicht wird auch als multidimensionaler Datenwürfel bezeichnet. [vgl. [35, 8, 68]]

4.3 Muster in Daten

Nachdem ein paar Beispiele zur Datenherkunft angeführt wurden, behandelt der folgende Abschnitt mögliche Muster, welche in Datenmengen gefunden werden können (prädiktiv oder deskriptiv).
Oftmals ist es schwierig Muster im Datenbestand zu finden, daher wäre es günstig, wenn Data-Mining Systeme Benutzer auf möglicherweise interessante Muster aufmerksam machten. Der Benutzer soll aber auch einem Data-Mining-System Hinweise geben können, welche Muster im Datenbestand für ihn vermehrt von Interesse sind. [vgl. [29, 35]]

Es können folgende Muster in Datenbeständen unterschieden werden [vgl. [73, 65, 8, 44]]:

- Assoziationen

  beschreiben Relationen oder Zusammenhänge im Datenbestand, die zwischen mehreren Objekten bestehen können. Es können zum Beispiel Einkaufswarenkörbe auf Assoziationen bezüglich der enthaltenen Produkte untersucht werden. Dabei kann ermittelt werden, ob ein Kunde, der beispielsweise Bier kaufte auch Wein im Einkaufskorb hatte. [vgl. [73]]

- Klassifikationen

  definieren in einem ersten Schritt Klassen beziehungsweise Kategorien und teilen danach Objekte in die zuvor festgelegten Klassen ein. Als Beispiel sei hier der Bereich Gesundheit erwähnt. Dabei können einzelne Krankheiten bezüglich ihres Heilerfolgs klassifiziert werden, mit: heilbar, teilweise heilbar oder unheilbar. [vgl. [65]]

- Sequenzen

  können auch als zeitabhängige Assoziationen gesehen und über einen gewissen Zeitraum betrachtet werden. Die zeitliche Abfolge von Ereignissen spielt dabei eine wichtige Rolle. [vgl. [8]]

- Cluster

  funktionieren ähnlich wie Kategorien bei der Klassifikation. Der Unterschied besteht darin, dass vor dem Prozessbeginn keine Kategorien definiert wurden. Die Objektkategorien werden schrittweise iterativ bestimmt. Jeder Cluster enthält daher nur Objekte, welche ähnliche Eigenschaften aufweisen. Beispielsweise können diverse Nahrungsmittelwarenkörbe analysiert und Cluster von Daten ermittelt werden: Kunden, welche nur Bio-Nahrung im Einkaufskorb hatten oder Luxuskunden, die vermehrt Kaviar und Lachs kauften. [vgl. [44]]

4.4 Wissensvorbereitung

Der Begriff Data-Mining beschreibt maschinelles Lernen mit Hilfe diverser Algorithmen. Maschinelles Lernen beruht dabei auf dem Verstehen und Lernen von Konzepten, Instanzen und Attributen. Dabei können Muster in Datenmengen gefunden werden. Algorithmen benötigen Wissen in Form von Daten als Eingabe, damit eine entsprechende Ausgabe generiert werden kann. Dieses Wissen muss vor der Verarbeitung vorbereitet werden.

4.4.1 Konzepte, Instanzen und Attribute

Ein Konzept beschreibt was gelernt werden soll, unabhängig von der Art des Lernens und des gewählten Algorithmus. Konzepte müssen nicht immer endgültig definiert sein. Klassifikation oder Clusterung stellen dabei Varianten dar, um Konzepte zu erlernen. Bei der Klassifikation werden Objekte anfangs manuell klassifiziert; diese Objekte werden als Lernschema für einen Algorithmus verwendet. Basierend auf dem Gelernten ist in weiterer Folge ein Algorithmus im Stande neue Objekte zu finden und zu klassifizieren. Beim Clustern von Daten wird im Gegensatz zum Klassifizieren einem Algorithmus anfangs kein Lernschema vorgegeben. Die Kategorisierung der Daten wird erst im Laufe des Prozesses erlernt.
Es kann zum Beispiel aus Wetterdaten ermittelt werden, unter welchen Wetterumständen ein

Fußballspiel stattfindet oder nicht. Für jedes Datenobjekt wird ermittelt, ob es zum Konzept beziehungsweise in die Klasse ”es findet ein Spiel statt” oder ”es findet kein Spiel statt” gehört. Ein Algorithmus erhält dabei als Eingabe ein Schema von Wetterdaten, welches als Trainingsdatenmenge zum Lernen dient. Nach der Lernphase werden dem Algorithmus die echten Wetterdaten zum Klassifizieren gegeben. Gleichzeitig liegen bereits fertig klassifizierte Ergebnisse derselben Datenmenge über die wahren Entscheidungen auf Grund der Wetterdaten vor. Sie wurden durch manuelle Klassifikation ermittelt. Mittels Vergleich der beiden klassifizierten Datenmengen kann die Genauigkeit und Qualität des maschinellen Lernvorganges verifiziert werden.
Die Eingabe beim maschinellen Lernen ist eine Menge von Daten, welche aus einzelnen Objekten besteht. Diese Objekte können wiederum klassifiziert, miteinander assoziiert oder in Cluster eingeteilt werden. Objekte werden auch Instanzen genannt und können aus mehreren vordefinierten Attributen bestehen. Die Anordnung der Instanzen mitsamt Attributen wird durch eine Matrix bestimmt. Jede Zeile der Matrix beschreibt dabei ein einzelnes Objekt und die Spalten jeder Zeile bezeichnen die Objektattribute mitsamt Ausprägungen. Diese Matrix kann auch mit einer Tabelle in einer Datenbank verglichen werden. Tabellen können maschinell relativ einfach verarbeitet werden.
Instanzen werden durch eine Vielzahl von Attributen charakterisiert, welche unterschiedliche Wertausprägungen besitzen können. Im nachfolgenden Abschnitt werden Attribute mit ihren möglichen Wertebereichen vorgestellt. [vgl. [75]]

4.4.2 Attribute und ihre Ausprägungen

Bevor mit dem Data-Mining-Prozess begonnen werden kann, ist es sinnvoll Überlegungen in Bezug auf Dateninhalte und -qualität anzustellen. Es gibt viele Eigenschaften, welche ein Objekt in einer Datenbanktabelle besitzen kann. Falls diese Eigenschaften in ihrer Bedeutung missverstanden werden, könnte das gegebenenfalls zu Problemen bei der Analyse des Datenbestandes führen. Daher werden diese Eigenschaften und ihre möglichen Wertausprägungen näher erklärt [vgl. [37, 10, 75]]:

- Nominale Werte

  können als Namen verstanden werden und bezeichnen beispielsweise Farben oder Gegenstände. Nominale Werte haben keinerlei Ordnung oder Beziehung zueinander. Ein Beispiel für die nominale Ausprägungen des Attributes Farbe wären Rot, Blau, Grün. Ebenso können Werte in nummerischer Form vorkommen; sie sind jedoch nur als Beschriftung des nummerischen Wertes zu verstehen und besitzen daher keine mathematische Bedeutung. Ein nominales Attribut kann daher bei einem Vergleich nur auf exakte Gleichheit geprüft werden. Im Bezug auf das Attribut Farbe bedeutet das, dass nur exakt eine oder keine der Ausprägungen darauf zutrifft. [vgl. [37, 10, 75]]

- Ordinale Werte

  sind ähnlich zu nominalen Werten zu verstehen. Der Unterschied besteht jedoch darin, dass ordinale Werte im Gegensatz zu nominalen Werten in einer sinnvollen Reihenfolge sortiert werden können. Es ist mitunter nicht immer sofort eindeutig, ob ein Attribut nominal oder ordinal ist. Beispielsweise kann die Höhe eines Gegenstandes mit den Werten klein, mittel oder groß auf Gleich- oder Ungleichheit geprüft werden. [vgl. [10]]

- Ganzzahlige Werte

  stellen gegensätzlich zu Nominalwerten den wahren Zahlenwert eines Attributes dar. In diesem Zusammenhang haben arithmetische Operationen sehr wohl eine mathematische Bedeutung. [vgl. [37], [10, 75]]

- Intervallwerte

  sind von nummerischer Natur und werden in gleichen Intervallschritten vom Nullpunkt oder Ursprung an gemessen. Ein Beispiel sind die Temperaturskalen Celsius und Fahrenheit. Mittels gemessener Werte kann daher die Aussage getätigt werden, ob eine Temperatur gemessen in Grad Celsius größer oder kleiner, als eine andere im gleichen Maß gemessene Temperatur ist. Es würde aber keinen Sinn ergeben, die gemessene Gradanzahl zu verdreifachen beziehungsweise zwei Intervallgrenzen miteinander zu addieren. Falls null Grad gemessen würde, wäre das Messcharakteristikum vorhanden, beträge jedoch null. [vgl. [37], [10, 75]]

- Verhältniswerte

  geben anders als bei Intervallwerten durch den Nullpunkt ein Nichtvorhandensein des Messcharakteristikums an. Auf das Beispiel mit den Temperaturen und die Maßeinheit Kelvin bezogen, hieße das, dass null Kelvin die kleinste, mögliche Temperatur, sprich, das Nichtvorhandensein von Temperatur angäbe. Verhältniswerte definieren einen absoluten Nullpunkt. Es können ebenso wie bei Intervallwerten mathematische Operationen darauf angewendet werden. [vgl. [37, 10, 75]]

Die vorgestellten Werttypen können vereinfacht in zwei Klassen eingeteilt werden, nämlich kategorisch und kontinuierlich. Kategorische Werte sind: nominal, binär und ordinal. Kontinuierliche Werte können sein: integer, Intervalle oder Verhältniswerte. [vgl. [10]]

4.4.3 Datenpräparierung

Nachdem die Attributarten erläutert wurden, sind diese für die eigentliche Verarbeitung zu präparieren. Der Grund dafür besteht deshalb, weil die rohen Daten meist von unterschiedlichen Quellen stammen und anfangs oft nicht die geeignete Form für eine weitere Verarbeitung besitzen.

Fehlerhafte Daten

Fehler in Rohdaten treten dabei sehr häufig auf. Sie können in jeweils zwei Kategorien eingeteilt werden: inhaltlich richtig oder inhaltlich falsch. Inhaltlich richtig bedeutet, dass ein Wert inhaltlich für ein bestimmtes Attribute möglich ist. Zum Beispiel sind bei integeren Attributen ganzzahlige Inhalte erlaubt. Ein inhaltlich falscher Wert für dieses Attribut wäre hingegen eine Zeichenkette bestehend aus Buchstaben.
Fehlerhafte Werte werden oft auch als Noise bezeichnet. Die Verwendung von Noise bezieht sich darauf, dass ein Wert für ein Attribut inhaltlich möglich ist, jedoch falsch erhoben wurde. Statt der Zahl 12.34 wurde die Zahl 1.234 erfasst. Inhaltlich gültige Fehler dieser Art sind problematisch, weil sie das Endergebnis verzerren können. Inhaltlich falsche Fehler, wie das Erfassen von ABCD statt der Zahl 12.34 wären weniger problematisch, da sie einfacher gefunden und behandelt werden können. In diesem Zusammenhang sei ebenso auf das Auftreten von statistischen Ausreißern bei gemessenen Datenwerten hingewiesen. Diese könnten ein Indiz für interessante Entdeckungen darstellen.
Fehler können in Datenmengen durch verschiedene Umstände verursacht werden. Gründe dafür

können sein: die fehlerhafte Aufzeichnung von Messungen, die missbräuchliche Verwendung von Messgeräten oder subjektive Entscheidungen. Dies führt dazu, dass eine nachgelagerte Analyse der Daten erschwert wird beziehungsweise, dass die dafür verwendeten Algorithmen (z. B. Klassifikation oder Clusteranalyse) schlichtweg falsche Ergebnisse liefern. [vgl. [57, 10]]

Fehlende Daten

Neben fehlerhaften sind fehlende Attributwerte zu beachten. Es könnte sein, dass für manche Attribute gar keine Werte erhoben wurden. Zum Beispiel kann sich eine bestimmte Art von medizinischen Daten entweder nur auf das männliche oder weibliche Geschlecht beziehen. In diesem Fall erscheint es sinnvoll die Daten nach dem Geschlecht aufzuteilen. Fehlende Daten können auch vorkommen, falls eine zuvor festgelegte Datenstruktur erweitert wird. In diesem Fall existieren für die neuen, zusätzlichen Felder der Struktur noch keine Datenwerte. Oftmals können fehlende Werte von Data-Mining-Algorithmen nicht ordnungsgemäß verarbeitet werden [vgl. [8, 10]]
Es gibt prinzipiell zwei unterschiedliche Ansätze im Umgang mit fehlenden Werten. Der eine Ansatz besagt, dass fehlende Werte immer ersetzt werden, der andere wiederum empfiehlt pro Datensatz eine Behandlung nach Bedarf. [vgl. [57]] Eine Variante, um fehlende Werte zu behandeln, wäre das Löschen von Instanzen, die zumindest einen fehlenden Wert enthalten. Die restlichen, übrigen Instanzen können für weitere Vorhaben verwendet werden. Ein Vorteil ist, dass bei dieser Methode keine neuen Datenfehler eingeführt werden. Nachteilig wirkt sich hingegen ein möglicher Verlust der Aussagekraft von vorhandenen Daten aus.
Eine andere Möglichkeit bietet das Ersetzen von fehlenden Werten mittels Durchschnittswert. Dabei ist wieder nach kategorischen und kontinuierlichen Attributen zu unterschieden. Für kategorische Attribute wird der am häufigsten vorkommende Wert als Durchschnittswert angenommen. Dieses Vorgehen eignet sich jedoch für nicht balancierte Wertverteilungen besser als für balancierte. Zum Beispiel besitzt ein Attribut Y die Werte k, l, m. Deren Verteilung auf das gesamte Attribut entspricht 88% für k, 7% für l und 3% für m. Es kann daher angenommen werden, dass die 2% der fehlenden Werte mit k zu besetzen sind. Ist die Verteilung der Werte k, l und m gleichmäßiger, wäre diese Annahme eher ungerechtfertigt. [vgl. [10]]
Für fehlende Werte bei kontinuierlichen Attributen wird häufig der Durchschnittswert der vorhandenen Attribute gebildet und als Ersatz verwendet.
Das Ersetzen von fehlenden Werten kann unter Umständen zu einer Verfälschung des Ergebnisses führen; dennoch hat dieser Effekt eher geringe Auswirkungen, falls die Anzahl der fehlenden Werte in Bezug auf die gesamte Datenmenge klein ist. Eine andere Technik um fehlende Datenwerte zu behandeln, stellt die Reduktion der Attribute dar. Je größer die Anzahl der Attribute pro Instanz ist, desto höher ist die Wahrscheinlichkeit, dass unnötige Information beim Analyseprozess verwendet wird. [vgl. [8], [10]]

4.5 Wissensdarstellung

Die Methoden zur Wissensvorbereitungen wurden im letzten Abschnitt erläutert. Dabei wurden Daten in Bezug auf ihre möglichen Inhalte charakterisiert. Bevor mit der Analyse der vorbereiteten Daten begonnen werden kann, muss noch die Frage der Wissensdarstellung geklärt werden. Daten können sehr vielfältig repräsentiert werden; zum Beispiel durch Regelnetzwerke, Entscheidungsbäume oder Tabellen. [vgl. [75]]

4.5.1 Tabellen

Die einfachste Möglichkeit vorhandenes Wissen strukturiert darzustellen, sind Tabellen. Diese Repräsentationsart kann sowohl für die Eingabe als auch für die Ausgabe von Daten beim maschinellen Lernen verwendet werden. Als Beispiel sei hier eine Tabelle mit dem Inhalt von

Kinoprogrammdaten erwähnt. Um zu entscheiden, ob ein Kinofilm besucht wird, werden die Bedingungen in der Tabelle analysiert und ausgewertet. Tabellen können für Prognosen verwendet werden. In diesem Fall werden sie als Regressionstabellen bezeichnet und im Bereich der nummerische Vorhersage angewendet. [vgl. [75]]

4.5.2 Bäume

Bäume stellen eine weitere Variante dar, um Wissen zu repräsentieren. Sie werden auch häufig mit dem Namen Entscheidungsbäume bezeichnet. In dieser baumähnlichen Struktur befinden sich Knoten. Die einzelnen Knoten testen jeweils verschiedene Attribute von konkreten Objekten. Jeder Pfad im Baum, von der Wurzel bis zu einem bestimmten Blatt, liefert ein Testergebnis. Die Blätter des Baumes stellen die Klasse dar, durch welche Objekte klassifiziert werden können.
Beim Testen von Knoten werden die Attribute der zu testenden Objekte mit Konstanten verglichen. Neue, unbekannte Objektinstanzen werden im Baum von der Wurzel beginnend zu den Blättern durchgereicht. Ist die Objektinstanz bei einem Blatt angelangt, wird sie durch die Klasse des Blattes klassifiziert.
Bei nominalen Knoten gibt es pro Attributausprägung einen Nachfolgerknoten im Baum. Diese Ausprägungsknoten bezeichnen alle für ein konkretes Attribut gültigen Werte. Ein Attribut muss daher nicht erneut getestet werden. Manchmal können Attributausprägungen auch in zwei Wertebereiche aufgeteilt werden. Der Knotentest ergibt in weiterer Folge, dass die Attributausprägung in einem der Wertebereiche liegt. Ein Attribut muss bei einem solchen Vorgehen unter Umständen mehrfach getestet werden, um zu einer endgültigen Entscheidung zu kommen. Handelt es sich um nummerische Attribute, bietet sich eine dreiwertige Aufteilung an. Dabei kann beispielsweise unterschieden werden, ob eine Attributausprägung kleiner, größer oder gleich einem definierten Wert ist. Nummerische Attribute können auch mehrere Male im Baum getestet werden. Es können daher für jeden Test andere Konstanten verwendet werden. Besitzen Objekte fehlende Attributwerte, besteht die Möglichkeit den populärsten Pfad im Baum zu ermitteln. Dieser Pfad ist jener, welcher am häufigsten Objekte kategorisiert hat. Instanzen mit fehlenden Attributen werden infolgedessen nach diesem Pfad klassifiziert. [vgl. [35, 75]]

4.5.3 Regeln

Regelwerke bieten eine andere Variante von Wissensdarstellung. Bei Regeln gibt es bestimmte Vorbedingungen, auch Prämissen genannt, die zu erfüllen sind. Je nach Auswertung verschiedener Prämissen, ergibt sich eine Konklusion. Prinzipiell werden die Vorbedingungen mit dem logischen UND-Operator verknüpft. Das bedeutet, dass alle Prämissen zutreffen müssen, damit aus ihnen die Konklusion gefolgert werden kann. In einzelnen Fällen, lassen sich die Vorbedingungen aber auch mit dem logischem ODER-Operator verketten. Die Auswertung von mit ODER verknüpften Bedingungen funktioniert so, dass nur eine der Prämissen zutreffen muss, damit auf ein Ergebnis geschlussfolgert werden kann. [vgl. [75]]

Klassifikationsregeln

Es wird davon ausgegangen, dass Objekte zuvor definierten Klassen zugeordnet werden. Die Zuordnung ist eindeutig. Es darf kein Objekt zwei oder gar mehr unterschiedlichen Kategorien zugeordnet worden sein. Beispielsweise könnte der Erfolg von Softwareprojekten mit den Begriffen ”erfolgreich umgesetzt”, ”umgesetzt” oder ”fehlgeschlagen” klassifiziert werden. Es könnte aber auch das Eintreten von Regenschauern in den kommenden Tagen mit ”sehr wahrscheinlich”, ”wahrscheinlich” oder ”unwahrscheinlich” vorhergesagt werden.
Entscheidungsbäume bieten eine Möglichkeit, um die Klassifikation von Daten vorzunehmen. Dabei können Klassifikationsregeln aus einem Entscheidungsbaum abgelesen werden. Für jedes

Blatt des Baumes existiert eine Regel. Die Knoten von der Wurzel des Baumes zu den jeweiligen Blättern enthalten die Bedingungen der Regeln. Ein konkretes Blatt selbst ist die Folgerung einer gewissen Regel und wird als Klasse bezeichnet. Die Abarbeitungsreihenfolge von Klassifikationsregeln ist nicht relevant. Sie kann frei bestimmt werden. [vgl. [75]] Eine Regel repräsentiert eine Art Ausschnitt des Entscheidungsbaumes. Es können ebenso neue Regeln hinzugefügt werden, ohne bereits existierende Regeln zu beeinträchtigen. Es ist jedoch zu beachten, dass eine isolierte Regelauswertungen falsche Klassifikationen hervorrufen kann. Das wäre zum Beispiel der Fall, falls zwei unterschiedliche Regeln unterschiedliche Schlussfolgerungen für ein und dieselbe Objektinstanz liefern würden. Bei einer isolierten Abarbeitung der Regeln sei daher Vorsicht geboten.
Bei der Klassifikation können generell zwei Arten unterschieden werden. Zum einen gibt es die Apriori-Klassifikation. Diese Vorgehensweise erzeugt Klassen ohne die vorhandenen Daten zu beachten. Zum anderen existiert die Aposteriori-Klassifikation. Hier wird davon ausgegangen, dass die Klassen auf Grund der Daten erzeugt werden. In diesem Zusammenhang sei auch auf die supervisierte Suche hingewiesen. Sie ermöglicht die Zuordnung von neuen Objekten zu bereits bestehenden Klassen mittels Apriori-Klassifikation. [vgl. [10],[32],[75]]

Assoziationsregeln

Die Analyse von Daten kann auf Grund von vermuteten Assoziationen erfolgen. Diese Beziehungen können zwischen ein oder mehreren Objektinstanzen in einer Datenmenge vorkommen. Assoziationsregeln funktionieren ähnlich wie die bereits erklärten Klassifikationsregeln. Der Unterschied besteht darin, dass Assoziationsregeln auch das Vorkommen von Objekten vorhersagen können und nicht nur die Klasse der Objekte. Zu den grundlegenden Data-Mining-Verfahren zählen hier der Apriori-Algorithmus, FP-growth und Eclat [vgl. [36]] Im Rahmen dieser Arbeit wird näher auf den Apriori-Algorithmus eingegangen.
Assoziationsregeln werden meist in der Form $X \rightarrow Y$ geschrieben. Das bedeutet: falls X auftritt, tritt auch immer Y auf. Dabei ist X wieder die Prämisse und Y wird als Schlussfolgerung bezeichnet. Die beiden Elemente wurden gemeinsam in den vorliegenden Daten gefunden.
Angenommen eine Datenmenge besteht aus N Transaktionen. Jede dieser Transaktionen beinhaltet diverse Elemente. X und Y sind zwei Elemente daraus, die gemeinsam in 15% Prozent der Transaktionen vorkommen. Diese Eigenschaft wird generell als Support bezeichnet. Und immer wenn X in einer Transaktion vorkommt besteht zu 75% die Chance, dass auch Y vorkommt. Dieses Merkmal heißt Konfidenz. Support und Konfidenz geben die Relevanz einer Assoziationsregel an. [vgl. [32, 37, 75]] Hohe Konfidenz sagt aus, dass die Regel genug oft wahr ist und rechtfertigt sie als Entscheidungsgrundlage. Hoher Support heißt, dass die Regel genug oft in der Datenmenge vorkommt und daher von Interesse sein könnte. [vgl. [44]] Support und Konfidenz können wie folgt berechnet werden.

Für den Support von X ergibt sich daher folgende Formel [vgl. [32]]:

$$Support(X) = \frac{Anzahl\ wie\ oft\ X\ vorkommt}{N} = P(X)$$

N ist die Anzahl der Transaktionen. Der Support von X wird berechnet indem das Vorkommen von X durch die Transaktionsanzahl N dividiert wird. Beim Support von X und Y funktioniert die Berechnung analog dazu [vgl. [32]]:

$$Support(XY) = \frac{Anzahl\ wie\ oft\ X\ und\ Y\ gemeinsam\ vorkommen}{N} = P(X \cap Y)$$

Die Konfidenz von $X \rightarrow Y$ errechnet sich folgendermaßen [vgl. [32]]:

$$Konfidenz(X \rightarrow Y) = \frac{Support(XY)}{Support(X)} = \frac{P(X \cap Y)}{P(X)} = P(Y|X)$$

$P(Y \mid X)$ gibt an, wie hoch die Wahrscheinlichkeit von Y ist, wenn X vorgekommen ist und wird als bedingte Wahrscheinlichkeit bezeichnet. [vgl. [37]]

4.5.4 Cluster

Cluster stellen neben der Klassifikation eine ähnliche Variante zum Kategorisieren von Objekten dar. Beim Clustering werden jedoch nicht einzelne Instanzen klassifiziert, sondern Objekte in sinnvollen Gruppen organisiert. Es wird daher eine kleine Anzahl von Clustern gebildet, anstatt einer großen Anzahl von einzeln klassifizierten Objekten. Cluster sind meist von unterschiedlicher Größe. Die Instanzen innerhalb eines Clusters sind von ihren Eigenschaften sehr ähnlich zueinander. Sie unterscheiden sich jedoch deutlich von Objekten in anderen, benachbarten Clustern. Die Darstellung von Clustern erfolgt meist zweidimensional.
Cluster sind in den meisten Fällen disjunkt zueinander. Spezielle Methoden zur Clusteranalyse können aber ein Objekt auch zu zwei Clustern zuordnen. In diesem Fall ist eine zweidimensionale Darstellung mit überlappenden Teilbereichen zu wählen – diese wird auch als Venn Diagramm bezeichnet.
Andere Verfahren berechnen mit welcher Wahrscheinlichkeit Objekte einem bestimmten Cluster zugehörig sind. Dies lässt sich gut in tabellarischer Form anzeigen.
Wiederum andere Algorithmen zum Clustern von Daten erzeugen hierarchische Strukturen. Dabei werden auf oberster Ebene Cluster erzeugt und daraus ergeben sich anschließend wieder ein oder mehrere Subcluster. Objekte, welche sich in Clustern auf unteren Ebenen befinden, sind ähnlicher zueinander als Objekt auf höheren Ebenen. Eine Repräsentationsform für diese Clusterart bieten so genannte Dendrogramme. [vgl. [32, 37, 75]]

5 Fallbeispiel Fuzzing-Werkzeug

Das in dieser Arbeit eingesetzte Framework geht im Wesentlichen nach dem im Abschnitt 3.3.1 beschriebenen Fuzzing-Prozess zum Testen von Software vor. In diesem Kapitel wird zuerst ein Überblick über die Architektur sowie Funktionsweise des konkreten Fuzzers gegeben, danach erfolgt die Veranschaulichung des verwendeten Datenmodells und die Beschreibung der Zusammenhänge durch die beispielhafte Erklärung von einigen Sicherheitsbedrohungen.
Der Fuzzer besteht aus verschiedenen Analyzern, welche Software nach unterschiedlichen Aspekten testen, die erzeugten Testergebnisse sammeln, und bewerten, ob im Zuge eines Testlaufs bei der zu testenden Software (SUT) möglicherweise Sicherheitsfehler aufgetreten sind oder nicht. Es werden beispielsweise Daten aus erhaltenen Antwortnachrichten und Log-Dateien extrahiert oder Screenshots von der graphischen Oberfläche der zu testenden Software abgezogen, falls bestimmte Farben (z. B. Rot) in Textausgaben am Bildschirm entdeckt worden sind. Die Analyzer bewerten ihre Resultate pro durchgeführtem Testfall und erzeugen hierfür Tabellen in einer Datenbank. [vgl. [70, 67]]
Der Fuzzer wird vor jedem Testlauf mit entsprechenden Parametern konfiguriert. Dabei kann beispielsweise eingestellt werden, welche Art von Applikation getestet wird, wie der Nachrichtenaustausch zwischen dem Fuzzer und der zu testenden Applikation stattfindet, mit wie vielen Testfällen der Fuzzer die Applikation testet oder welche Priorität die einzelnen Analyzerergebnisse im Bewertungsprozess von Sicherheitsfehlern besitzen.
Nachdem der Fuzzer konfiguriert wurde, beginnt der erste Testlauf gegen eine zuvor festgelegte Applikation.

5.1 Architektur des Fuzzers

Das Fuzzing-Framework bietet die Möglichkeit unterschiedliche Applikationen zu testen, zum Beispiel das Erkennen von Sicherheitslücken in SIP-basierten Softphones (z. B. QuteCom). [vgl. [67]] Es baut auf einem generischen Ansatz auf und benützt im Rahmen von Sicherheitstests Nachrichtenvorlagen (Templates) zur Testgenerierung. In Abbildung 5.1 wird eine exemplarische Darstellung des Frameworks mit dem zu testenden Softwaresystem (SUT) gezeigt.

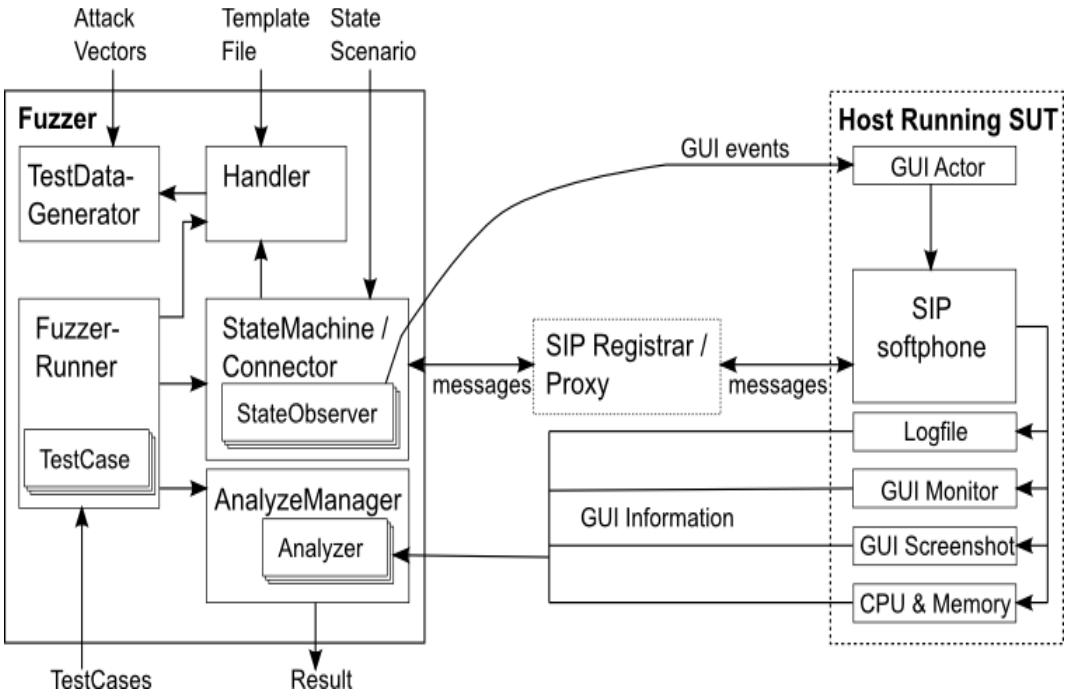

Abbildung 5.1: Schematische Darstellung der Architektur des Fuzzing-Werkzeugs [vgl. [67]]

Der Fuzzer besitzt eine Komponente StateMachine, welche mit dem zu testenden System über den GUI Actor sowie GUI Monitor kommuniziert. Dabei schickt die StateMachine Ereignisse an den GUI Actor, welcher die GUI des zu testenden Softphones bedient. Der GUI Actor ist ebenso wie der GUI Monitor auf dem SUT installiert. Der GUI Montitor überwacht das zu testende System permanent und versorgt die Analyzer des Fuzzers mit verschiedenen Informationen, welche vom SUT produziert wurden (z. B. Fehlermeldungen, Bildschirmausgaben usw.). Auf dem zu testenden System befinden sich des Weiteren noch verschiedene vom SUT produzierte Log-Dateien, eine Screenshot-Komponente sowie eine CPU- und Speicherüberwachung.
Der Fuzzer wurde mit einem SIP-Plugin und verschiedenen Analyzern erweitert, welche im Abschnitt 5.2 näher beschrieben werden. Zunächst folgt eine genauere Beschreibung des Fuzzing-Werkzeugs.
Wie in Abbildung 5.1 dargestellt, besteht der Fuzzer aus den Komponenten FuzzerRunner, TestDataGenerator, Handler, StateMachine und AnalyzerManager (inkl. Analyzern). Der FuzzerRunner ist für die Ausführung der Testläufe zuständig und kontrolliert jeweils die anderen, erwähnten Komponenten. Der Handler ist für die Erzeugung von Nachrichten verantwortlich, welche mittels Connector zum SUT geschickt werden. Jener Connector ist faktisch in ständiger Kommunikation mit dem zu testenden System. Der TestDataGenerator erzeugt randomisiert oder aus vordefinierten Attack-Vektordateien Testwerte, mit denen das SUT getestet wird. Der AnalyzerManager erhält von seinen Analyzern Testresultate. Auf Grund dieser bestimmt er mit welcher Wahrscheinlichkeit die einzelnen Testfälle Sicherheitsfehler beim SUT hervorgerufen haben könnten.
Mit Hilfe der StateMachine kann der Fuzzer das zu testende System vor beziehungsweise nach einem Testfall oder gesamten Testlauf präparieren. Dies wird in Abbildung 5.2 am Beispiel von SIP veranschaulicht.

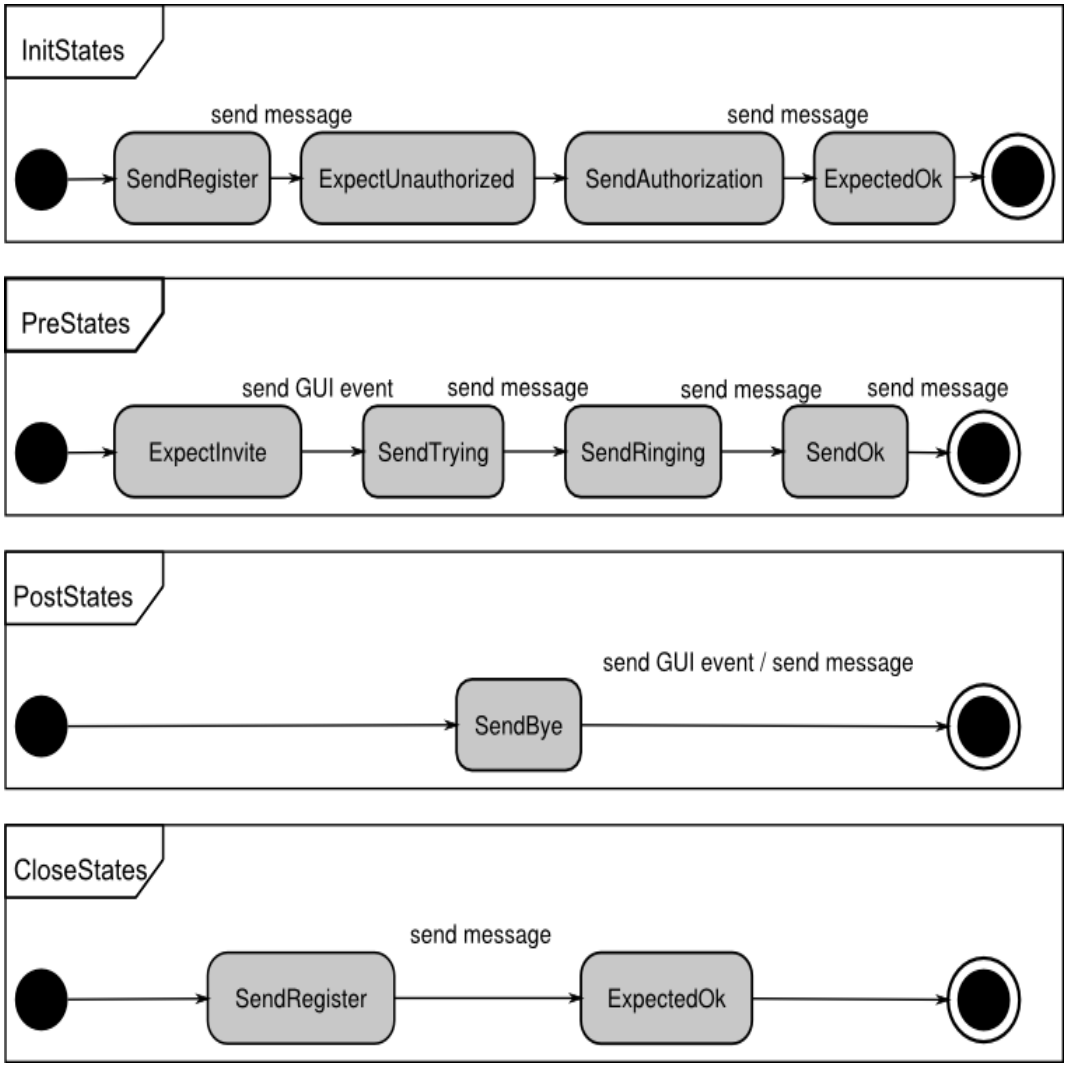

Abbildung 5.2: Schematische Darstellung der StateMachine-Phasen des Fuzzers beim Testen von SIP-basierten Softphones [vgl. [67]]

Die Komponente StateMachine ist ein Teil der Komponente Connector und kommuniziert entweder direkt oder via Proxy mit dem zu testenden System. Das Testen von Softphones basiert auf dem im Abschnitt 3.1.1 erwähnten Testen mit Transitionsgraphen. Dabei wird die Software mittels StateMachine in eine bestimmte Ausgangssituation (state) gebracht (z. B. Anmeldung beim Softphone). Es kann aber auch Aufräumarbeit (clean up) nach einem Test per StateMachine durchgeführt werden (z. B. Abmeldung vom Softphone). Die StateMachine besteht aus einem StateObserver, welcher es ermöglicht, beispielsweise mit der GUI des Softphones vor und nach verschiedenen Status zu interagieren. In Abbildung 5.2 werden verschiedene Status-Szenarien dargestellt. Im Rahmen eines Testlaufs meldet sich der Fuzzer bei der zu testenden Applikation (SUT) an und führt dazu InitStates aus. Danach wird für jeden Testfall PreStates davor und PostStates danach ausgeführt. Nach dem gesamten Testlauf wird schließlich CloseStates ausgeführt.

5.2 Datenbankstruktur des Fuzzing-Werkzeugs

Wie schon im vorangegangen Abschnitt erklärt, besteht der Fuzzer aus einem AnalyzeManager. Dieser beinhaltet unterschiedliche Analyzer, die innerhalb eines Testlaufs das Verhalten der zu

testenden Applikation analysieren und Auffälligkeiten pro Testfall in einer Datenbank vermerken. Diese enthält pro Analyzer mindestens eine Tabelle. Abbildung 5.3 veranschaulicht das Datenmodell des Fuzzing-Frameworks. Nachfolgend werden die in der Datenbank vorhandenen Analyzer-Tabellen beschrieben, um deren Inhalt zu verdeutlichen [vgl. Abbildung 5.3]:

- FILEREADER_ANAYLZER

  In Tabelle 5.1 befinden sich pro durchgeführtem Testfall die Gewichtung des Analyzers, das vorläufige Resultat des Analyzers (also mit welcher Wahrscheinlichkeit vermutet wird, dass es sich um einen Sicherheitsfehler handelt) und die Antwortnachricht der zu testenden Applikation.

Spalte	Beschreibung
TC_ID	Nummer des Testfalls
PRIORITY	Gewichtung des Analyzers
RESULT	vorläufige Testfallbewertung
ANALYZER_LOG	Log-Eintrag zum Testfall des Analyzers

Tabelle 5.1: Beschreibung der Tabelle FILEREADER_ANALYZER [vgl. [67]]

- FILEREADER_ANALYZER_KEYWORDS

  Tabelle 5.2 beschreibt welche Fehlermeldungen in der Antwortnachricht der zu testenden Applikation gefunden werden konnten und ob darin zum Beispiel Schlüsselwörter, wie ”Eingabefehler”, ”Formatfehler”, ”Ausnahmefehler”, ”Sicherheitsfehler”, ”Parsefehler” usw. entdeckt worden sind.

Spalte	Beschreibung
TC_ID	Nummer des Testfalls
KEYWORDS	die vom Analyzer gefundenen Fehlermeldungen
FROM_FILE	neue Fehlermeldung (ja/nein)
RATING	Bewertung der gefundenen Fehlermeldungen

Tabelle 5.2: Beschreibung der Tabelle FILEREADER_ANALYZER_KEYWORDS [vgl. [67]]

- RESPONSE_ANALYZER

  In Tabelle 5.3 wird pro Testfall die Länge der erhaltenen Antwortnachricht von der zu testenden Applikation, die Antwortnachricht selbst sowie deren Längendifferenz zu gängigen Antwortnachrichten des SUT gespeichert. Außerdem gibt es wieder die bereits erwähnte Gewichtung des Analyzers. Die vorläufige Bewertung gibt an, ob es sich bei den Testfällen um potenzielle Sicherheitsfehler handelt oder nicht.

Spalte	Beschreibung
TC_ID	Nummer des Testfalls
LENGTH	Länge der Nachricht
LENGTH_DIFFERENCE	Differenz zur gängigen Nachricht
PRIORITY	Gewichtung des Analyzers
RESULT	vorläufige Testfallbewertung
ANALYZER_LOG	Log-Eintrag zum Testfall des Analyzers

Tabelle 5.3: Beschreibung der Tabelle RESPONSE_ANALYZER [vgl. [67]]

- RESPONSE_CLASSSIFICATION_ANALYZER

  Tabelle 5.4 beinhaltet die Klassifikation der Antwortnachrichten. Dieser Analyzer berechnet im Gegensatz zum ResponseAnalyzer nicht die Länge der Antwortnachricht, sondern klassifiziert Antworten nach ihrem Inhalt und teilt sie in entsprechende Kategorien ein.

Spalte	Beschreibung
TC_ID	Nummer des Testfalls
VARIATION	Einstufung der Nachricht zu einem initial vorhandenen Wert beim Starten des Testlaufs
VARIATION_DIFFERENCE	Differenz zur Nachricht zu einem initial vorhandenen Wert beim Starten des Testlaufs
PRIORITY	Gewichtung des Analyzers
RESULT	vorläufige Testfallbewertung
ANALYZER_LOG	Log-Eintrag zum Testfall des Analyzers

Tabelle 5.4: Beschreibung der Tabelle RESPONSE_CLASSSIFICATION_ANALYZER [vgl. [67]]

- RESPONSETIME_ANALYZER

  Tabelle 5.5 enthält jene Werte, welche der Fuzzer in Bezug auf die Antwortzeiten des SUT gemessen hat. Außerdem wird noch festgehalten, wie groß die Differenz zur gängigen Antwortzeit des SUT war. Weicht die Antwortzeit eines Testfalls gravierend von den gängigen Antwortzeiten ab, wird von diesem Analyzer (analog zu den anderen) ein möglicher Sicherheitsfehler vermutet.

Spalte	Beschreibung
TC_ID	Nummer des Testfalls
DURATION	Dauer der Antwort
DURATION_DIFFERENCE	Differenz zur gängigen Antwortdauer
PRIORITY	Gewichtung des Analyzers
RESULT	vorläufige Testfallbewertung
ANALYZER_LOG	Log-Eintrag zum Testfall des Analyzers

Tabelle 5.5: Beschreibung der Tabelle RESPONSETIME_ANALYZER [vgl. [67]]

- CPU_LOAD_ANALYZER

  In Tabelle 5.6 wird die CPU-Auslastung der zu testenden Applikation vom Fuzzer analysiert und pro Testfall gespeichert. Dieser Analyzer kann allerdings nur eingesetzt werden, wenn der Fuzzer die CPU-Auslastung mittels Überwachungskomponente [vgl. Abbildung 5.1] am SUT messen kann.

Spalte	Beschreibung
TC_ID	Nummer des Testfalls
LOAD	CPU-Auslastung

Tabelle 5.6: Beschreibung der Tabelle CPU_LOAD_ANALYZER [vgl. [67]]

- MEMORY_LOAD_ANALYZER

  Die Tabelle 5.7 enthält das vom Fuzzer beobachtete Speicherverhalten der zu testenden Applikation nach jedem Testfall. Die Integration und Verwendung dieses Analyzers zur Messung der Speicherauslastung verhält sich analog zum CPU_LOAD_ANALYZER.

Spalte	Beschreibung
TC_ID	Nummer des Testfalls
LOAD	Speicherauslastung

Tabelle 5.7: Beschreibung der Tabelle MEMORY_LOAD_ANALYZER [vgl. [67]]

- TCPPORT_ANALYZER

  In der Tabelle 5.8 wird festgehalten, ob der TCP-Port von der zu testenden Applikationen nach jedem durchgeführten Testfall verfügbar ist. Falls nicht, könnte das auf einen Systemabsturz hindeuten.

Spalte	Beschreibung
TC_ID	Nummer des Testfalls
REACHABLE	Erreichbarkeit des Ports (ja/nein)

Tabelle 5.8: Beschreibung der Tabelle TCPPORT_ANALYZER [vgl. [67]]

Der Fuzzer generiert neben den bereits vorgestellen Tabellen auch noch die beiden Tabellen ANALYZER_RESULT und TESTCASE_VARIATION. Sie stellen die Zusammenfassung aller Analyzerergebnisse pro einzelnem Testfall dar (ANALYZER_RESULT). Die Tabelle 5.9 veranschaulicht die Struktur der Tabelle ANALYZER_RESULT.

Spalte	Beschreibung
TC_ID	Nummer des Testfalls
ANALYZER_NAME	Name des Analyzers
ANALYZER_VALUE	Funde des Analyzers
ANALYZER_VALUE_DIFFERENCE	Differenzen zu einem konfigurierten Wert, welcher beim Start des Testlaufs festgelegt wird
RESULT	vorläufige Testfallbewertung durch den Analyzer
ANALYZER_LOG	Log-Eintrag zum Testfall des Analyzers

Tabelle 5.9: Beschreibung der Tabelle ANALYZER_RESULT [vgl. [67]]

Es wird ebenso ein Überblick über die durchgeführten Testfälle mit der finalen Ergebnisbewertung des Fuzzers gegeben (TESTCASE_VARIATION), was in Tabelle 5.10 dargestellt wird.

Spalte	Beschreibung
TC_ID	Nummer des Testfalls
ATTACK	Attack-Vektor (kompromittierende Attacke)
REQUEST	gesendete Anfrage an das SUT
RESPONSE	Antwort des SUT auf die Anfrage
ATTACK_GENERATOR	Komponente, welche den Attack-Vektor erzeugt hat
TEST_TYPE_NAME	Typ des Testfalls
TEST_TYPE_DESCRIPTION	Beschreibung des Testfalls
FINAL_RESULT	Endergebnis der Fuzzer-Bewertung
IS_VALID	Testfall aus der Initialphase (ja/nein)
RUNDURATION	Laufzeit des Testfalls

Tabelle 5.10: Beschreibung der Tabelle TESTCASE_VARIATION [vgl. [67]]

Im folgenden Abschnitt wird in Form von Beispielen erklärt, welche Zusammenhänge zwischen der Architektur des Fuzzers, dem Datenmodell und der im Kapitel 7 durgeführten Analyse bestehen.

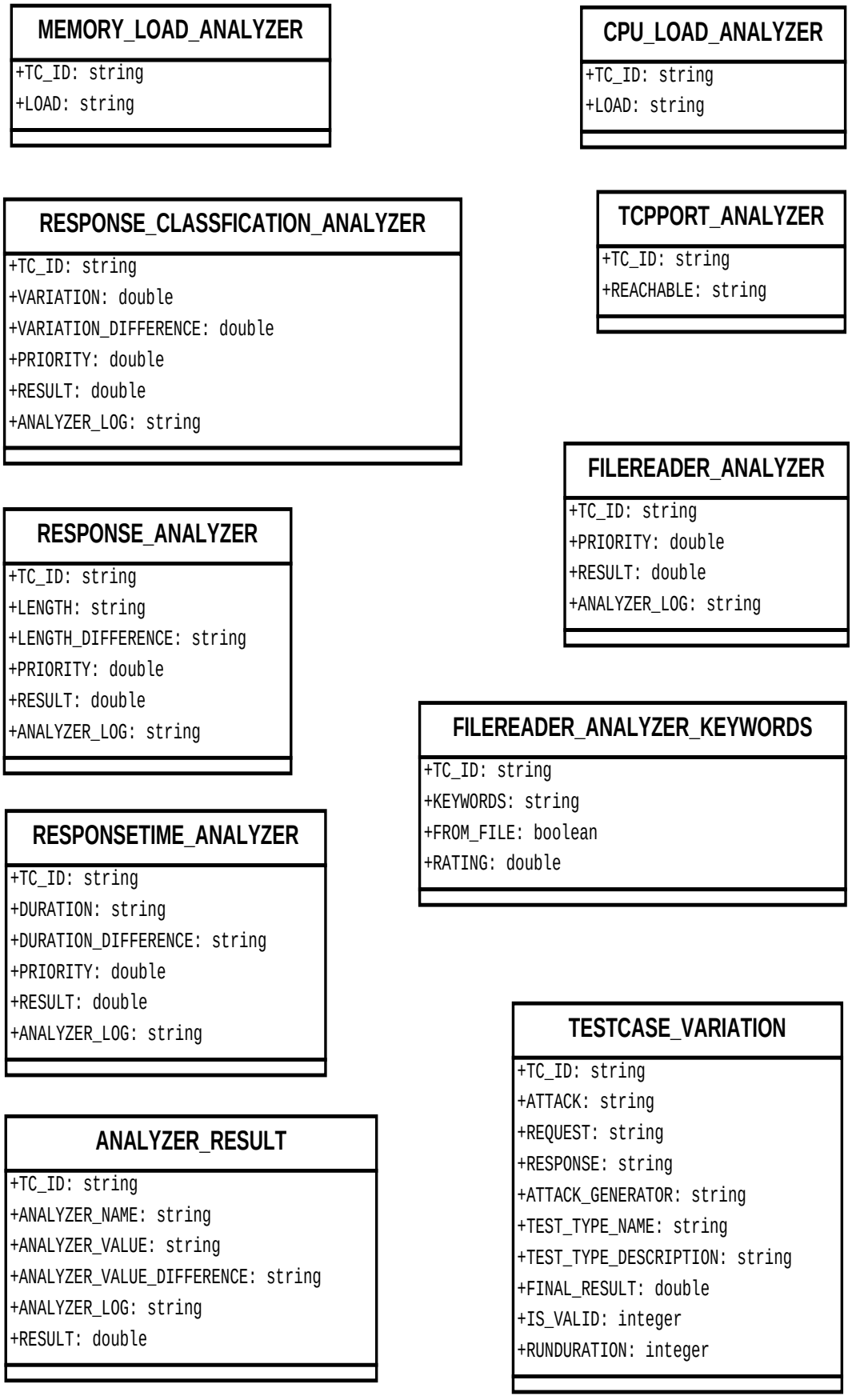

Abbildung 5.3: Exemplarisches Datenmodell des Fuzzing-Werkzeugs[vgl. [67]]

5.3 Sicherheitsbedrohnungen am Fallbeispiel des Fuzzing-Werkzeugs

Ziel des Fuzzing-Prozesses ist es Sicherheitsfehler in Software zu finden. Solche Fehler können durch so genannte Sicherheitsbedrohungen ausgelöst werden, beispielsweise durch Hacker. Bedingt durch Sicherheitsbedrohungen können unter Umständen sensible Daten in unbefugte Hände gelangen (z. B. Name, Geburtstdatum, Adresse usw.). [vgl. [51, 22]]
Um dies im Vorfeld so gut als möglich zu verhindern, kann beispielsweise ein Penetrationtest mittels Fuzzing eingesetzt werden, im Rahmen dessen potenzielle Sicherheitsbedrohungen erzeugt werden. Damit kann Software auf Sicherheitslücken überprüft werden, bevor Angreifer diese entdecken und benützen, um Schaden anzurichten.
Es gibt viele verschiedene Sicherheitsbedrohungen. In dieser Arbeit wird jedoch nur ein Teil davon erläutert, weil die Erklärung aller Sicherheitsbedrohungen den Rahmen sprengen würde. Daher wird ein schematischer Einblick in die Funktionsweise des konkreten Fuzzers gegeben. [vgl. [67]] Dies erfolgt anhand von Beispielen zu den Sicherheitsbedrohungen SQL-Injection, Pufferüberlauf und Command-Injection. Dieses Wissen wird nachfolgend im Kapitel 7 zum besseren Verständnis der Analysearbeit vorausgesetzt.

SQL-Injection

Eine große Sicherheitsbedrohung stellt in der heutigen Zeit SQL-Injection dar, wodurch Systeme kompromittiert werden können. SQL-Injection ist hierbei nicht auf eine bestimmte Datenbankart beschränkt.
Verschiedene Softwareysteme können davon betroffen sein, wie zum Beispiel die von Unternehmen. Sobald in jenen Applikationen eine SQL-Injection-Schwachstelle vorhanden ist, sind die Daten der Software in Gefahr, jedoch nur, falls der Angreifer die nötigen Zugriffsberechtigungen dazu besitzt. [vgl. [42]]
Ein erfolgreicher Angriff mittels SQL-Injection kann daher verschiedene Folgen haben. Je nach Erfahrung und Fertigkeit des Angreifers kann dieser einen mehr oder weniger gefährlichen Angriff starten, wie zum Beispiel das Auslesen von bestimmten in der Datenbank vorhandenen Tabelleninhalten. [vgl. [51]]
Der Vorgang zum Ausführen einer SQL-Injection ist relativ trivial: Ein potenzieller Angreifer schickt einen gefährlichen Eingabewert an die zu kompromittierende Software. Mit dieser Eingabe und durch Stringverkettung wird anschließend eine SQL-Abfrage generiert, an den Datenbankserver geschickt und versucht diesen zum Absturz zu bringen. [vgl. [42]] In Bezug auf den Fuzzer bedeutet das, dass im Rahmen von Sicherheitstests eine Software auf jene Schwachstelle untersucht werden kann. [vgl. Abschnitt 7] Ein Beispiel für eine SQL-Abfrage ist in Listing 5.1 gegeben.

```
1 SELECT * FROM Tabelle WHERE Tabelle.Attribut = 'Wert';
```

Listing 5.1: Exemplarisches Beispiel einer SQL-Abfrage [vgl. [51]]

Ein Angreifer könnte mit der Konkatenation von "' OR 1=1" (falls er 'Wert' kontrollieren kann) zu folgender Abfrage, wie in Listing 5.2 gelangen:

```
1 SELECT * FROM Tabelle WHERE Tabelle.Attribut = '' OR 1=1;
```

Listing 5.2: Präparierte Abfrage zur Erzeugung einer SQL-Injection [vgl. [51]]

SQL-Injection tritt durch fehlende Validierung der Eingabedaten auf, welche anschließend direkt zur Erstellung einer SQL-Abfrage verwendet werden. Die Abfrage 5.2 würde auf Grund der OR-Bedingung sämtliche Datensätze aus der Datenbanktabelle des SUT retournieren, falls der Angreifer die nötigen Zugriffsberechtigungen hat. Mit Hilfe eines Fuzzers kann das soeben erklärte Szenario nachgebaut, das SUT damit automatisiert getestet und eine eventuell vorhandene Schwachstelle behoben werden, bevor ein möglicher Angreifer Schaden anrichtet. Nachfolgend wird der schematische Ablauf eines Testfalls mit einer SQL-Injection-Attacke in Abbildung 5.4 veranschaulicht.

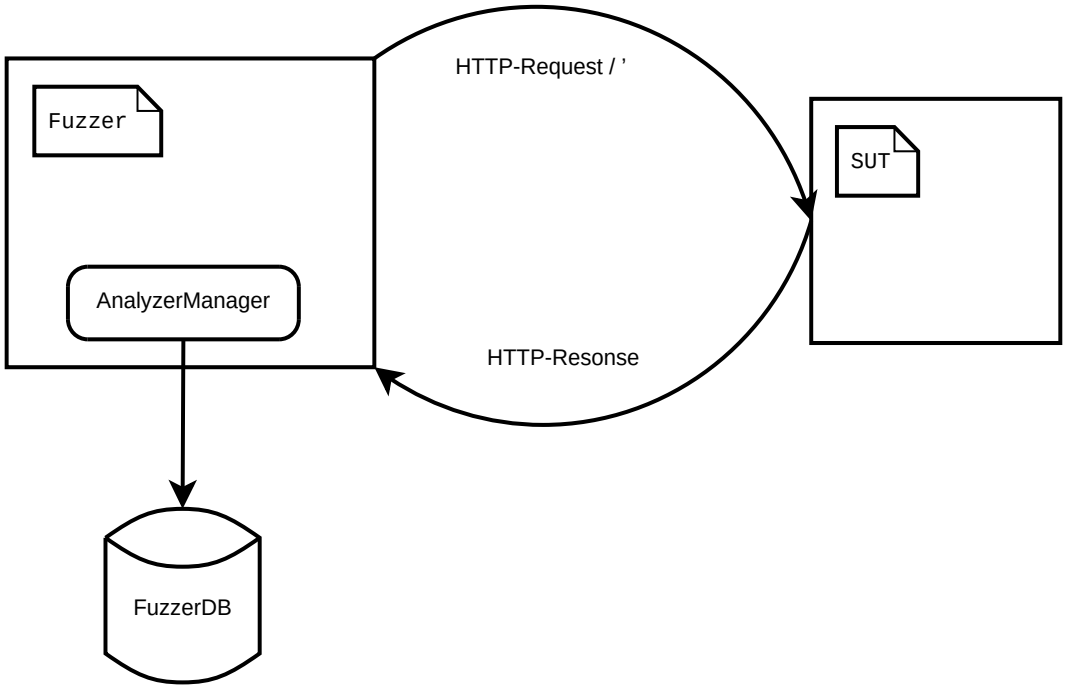

Abbildung 5.4: Schematischer Ablauf einer SQL-Injection während eines Fuzzing-Testlaufs [vgl. [67]]

Der Fuzzer schickt einen HTTP-Request an das zu testende System mit dem Eingabewert " ' ". Dieser Request wird in Abbildung 5.4 und zusätzlich durch das Codebeispiel in Listing 5.3 veranschaulicht.

```
1 GET /products/search?q=honig&n=' HTTP/1.1
2 Host: 192.168.0.1:80
3 Accept: application/json
4 Connection: Keep-Alive
5 Content-Type: application/x-www-form-urlencoded
6 Accept: application/json, text/javascript, */*; q=0.01
7 X-Session: ruOtxzBKtOUG=
8 X-Requested-With: XMLHttpRequest
```

Listing 5.3: Darstellung einer SQL-Injection durch den Fuzzer in Form eines HTTP-Headers [vgl. [67]]

Das SUT verarbeitet die Abfrage und liefert an den Fuzzer eine HTTP-Response, welche in Abbildung 5.4 und durch das Codebeispiel in Listing 5.4 dargestellt wird. Durch das Einfügen eines " ' " wird ein Fehler in der SQL-Abfrage erzeugt. Dies kann durch das Fuzzing-Werkzeug automatisch erkannt werden.

```
1 HTTP/1.1 500 Internal Server Error
2 Connection: keep-alive
3 Date: Mon, 07 Nov 2011 11:42:26 GMT
4 Content-Length: 114
5 Content-Type: text/html; charset=ISO-8859-1
6
7 Unexpected exception in handler: 'internal error in worker- ↩
    numberformatexception'
```

Listing 5.4: Darstellung einer Antwortnachricht des SUT auf eine SQL-Injection des Fuzzers in Form eines HTTP-Headers[vgl. [67]]

Diese HTTP-Response wird von den einzelnen Analyzern des AnalyzerManagers ausgewertet und in eine Datenbank gespeichert. Dabei würde der Analyzer zur Erkennung von Keywords zum Beispiel die Schlüsselwörter "internal", "error", "exception" und "numberformatexception" finden, diese bewerten und in seiner Datenbanktabelle speichern.
Der ResponseAnalyzer würde beispielsweise die Länge der HTTP-Response berechnen und diese analog zum Analyzer für Keywords in die zugehörige Tabelle persistieren.
Generell gilt: Es sollte bei einer stattgefundenen SQL-Injection so wenig Information wie möglich vom Softwaresystem nach außen zum Angreifer durchdringen. [vgl. [42, 51]]

Pufferüberlauf (Buffer-Overflow)

Neben SQL-Injection zählen Pufferüberläufe zu den bekanntesten Sicherheitsbedrohungen von Softwaresystemen. Pufferüberläufe können jeweils auftreten, falls versucht wird eine Menge von Daten in einen Puffer oder Speicher zu schreiben, welcher aber zu klein ist, um dieses Volumen zu fassen. Ein Beispiel nach van der Linden veranschaulicht wie Pufferüberläufe den reibungslosen Ablauf von Software beeinträchtigen können. [vgl. [51]] Angenommen ein Puffer hätte im Hauptspeicher eines Rechners die Gestalt wie in Tabelle 5.11.

X	X	X	Y	Y	Y	Y	Z	Z	Z
3	8	1	0	0	0	0	'b'	'e'	'n'

Tabelle 5.11: Hauptspeicheransicht eines Rechners vor der Erzeugung eines Pufferüberlaufs [vgl. [51]]

Bei einem Programmstart werden die drei Variablen X, Y und Z reserviert. X und Z werden mit Werten belegt, Y bleibt vorerst noch leer und wird zur Laufzeit des Programms mit Eingabewerten vom Benutzer befüllt. Ein potenzieller Angreifer schickt eine zu lange Zeichenkette als Eingabewert für die Variable Y, wie beispielsweise '@@@@@@@'. Falls das Programm die Länge der Zeichenkette nicht überprüft und sofort speichert, würde das den Speicher der Variablen Z überschreiben, sofern sich die Speicherstellen der Variablen Y und Z im Hauptspeicher nebeneinander befänden. Der Speicher des Rechners könnte daher nach einem erfolgten Pufferüberlauf wie in Tabelle 5.12 aussehen. Eine derartige Speicherkonstellation könnte unter Umständen ein Fehlverhalten im Zuge des weiteren Programmablaufs verursachen und das System zum Absturz bringen.

X	X	X	Y	Y	Y	Y	Z	Z	Z
3	8	1	’@’	’@’	’@’	’@’	’@’	’@’	’@’

Tabelle 5.12: Hauptspeicheransicht eines Rechners nach der Erzeugung eines Pufferüberlaufs [vgl. [51]]

Pufferüberläufe können generell Folgen von simplen Systemabstürzen bis hin zum Erhalt von Zugriffsberechtigungen auf Administratorebene führen.[vgl. [42, 51]] Fuzzing kann ebenso zur Überprüfung von Pufferüberlaufschwächen einer Software verwendet werden. Das aktuelle Fuzzing-Werkzeug bietet die Möglichkeit solche Tests durchzuführen. Es kann im Rahmen eines Testlaufs ein SOAP-Request an die zu testende Applikation gesendet werden, welches beispielsweise eine lange Zeichenkette enthält. Das wird in Abbildung 5.5 gezeigt. Falls die Länge der Zeichenkette vom SUT nicht überprüft wird und dieses nicht innerhalb des dargestellten Timeouts antwortet, kann angenommen werden, dass es beim Überschreiben eines Speicherbereichs zu einem Systemabsturz gekommen ist.

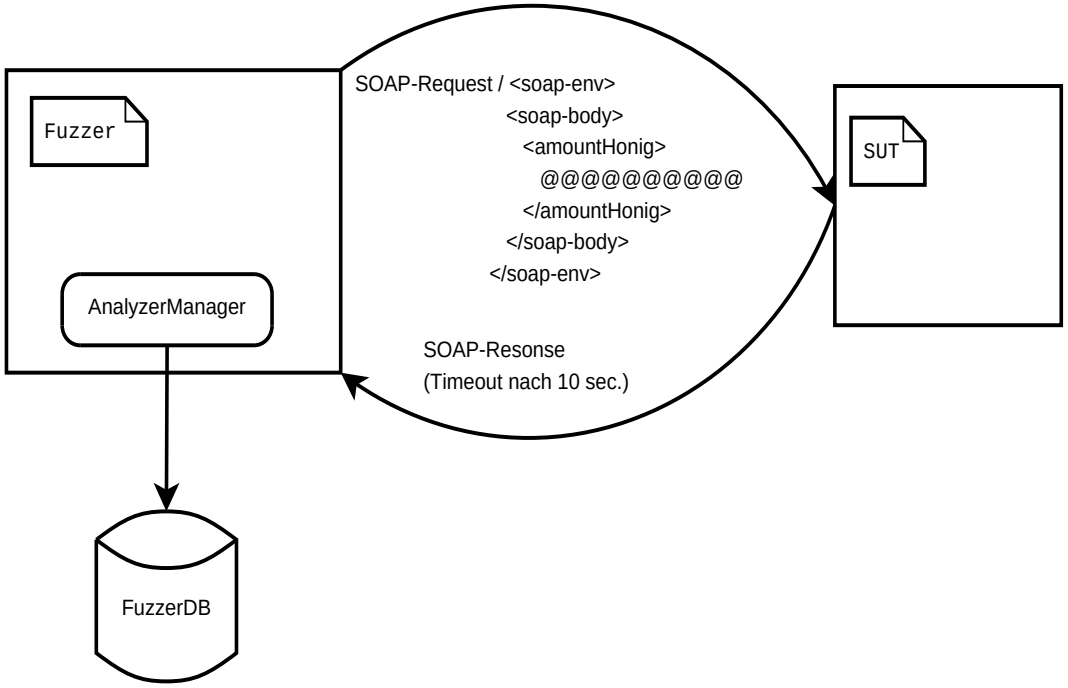

Abbildung 5.5: Schematischer Ablauf einer Character-Injection durch den Fuzzer während eines Testlaufs [vgl. [67]]

Ein entsprechender SOAP-Request zur Erzeugung eines Pufferüberlaufs wird in Listing 5.5 exemplarisch dargestellt. Dabei wird eine Folge von At-Zeichen ('@') zum SUT gesendet.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/↩
  encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <setHonigInLiter>
      <idHonigTopf xsi:type="xsd:long">3223</idHonigTopf>
      <amountHonig xsi:type="xsd:string">@@@@@@@@@@</amountHonig>
      <ipAddress xsi:type="xsd:string">127.0.0.1</ipAddress>
    </setHonigInLiter>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 5.5: Darstellung einer Character-Injection durch den Fuzzer in Form eines SOAP-Requests [vgl. [67]]

Nachdem die Abarbeitung des SOAP-Requests erfolgt ist, erhält der Fuzzer eine Antwortnachricht im SOAP-Format, welche beispielhaft in Listing 5.6 gezeigt wird.

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/↩
  envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchemainstance">
  <soapenv:Body>
    <soapenv:Fault>
      <faultcode>soapenv:Server.userException</faultcode>
      <faultstring>java.lang.NumberFormatException: For input ↩
        string: \&quot;@@@@@@@@@@\&quot;</faultstring>
    </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 5.6: Darstellung einer Antwortnachricht des SUT in Form einer SOAP-Response auf eine Character-Injection des Fuzzers [vgl. [67]]

Es konnten hierbei zum Beispiel vom FileReaderAnalyzer im SOAP-Response aus Listing 5.6 Fehler gefunden werden, nämlich "NumberFormatException" und "userException". Dieser Analyzer und auch andere speichern hier wieder analog zum vorigen Beispiel die Bewertungen eines derartigen Testfallergebnisses in ihre Tabellen.

Command-Injection

Eine relativ ähnliche Sicherheitsbedrohung wie SQL-Injection stellt Command-Injection dar. Dies geschieht, falls bei Softwaresystemen Daten eingegeben werden, welche nicht bloße Datenwerte, sondern zum Beispiel Systemaufrufe von anderen Programmen sind. Angreifer können damit versuchen nicht erlaubten Zugriff auf Daten und andere Programme zu erhalten.

Angenommen ein Programm hätte die Gestalt wie in Listing 5.7.

```
1 char puffer[512];
2 sprintf(puffer,''system cat %s'', eingabe);
3 system(buffer);
```

Listing 5.7: Schematische Darstellung einer Command-Injection [vgl. [42]]

Falls ein Angreifer in den Puffer aus Listing 5.7 zum Beispiel die Zeichenkette ";ls -la;" eingibt, werden beide Befehle ausgeführt. Das Kommando "cat" erhält kein Argument, durch ";" wird die Eingabe des Kommandos beendet. Danach folgt die Ausführung des zweiten Kommandos, welches den Inhalt des aktuellen Verzeichnisses auf dem Rechner ausgibt, aber nur falls der Prozess des Programms die Rechte dazu besitzt. Die Terminierung des zweiten Kommandos endet analog zum ersten mit ";". [vgl. [42]] Mittels Command-Injection kann auch Shellcode zur Ausführung gebracht werden. [vgl. [20]]
Auch hier bietet Fuzzing eine Möglichkeit Software auf Sicherheitsgefährdungen dieser Art zu testen. Ein schematisches Beispiel wird dazu in Abbildung 5.6 dargestellt. Hierbei schickt der Fuzzer eine Command-Injection an die zu testende Software (SUT).

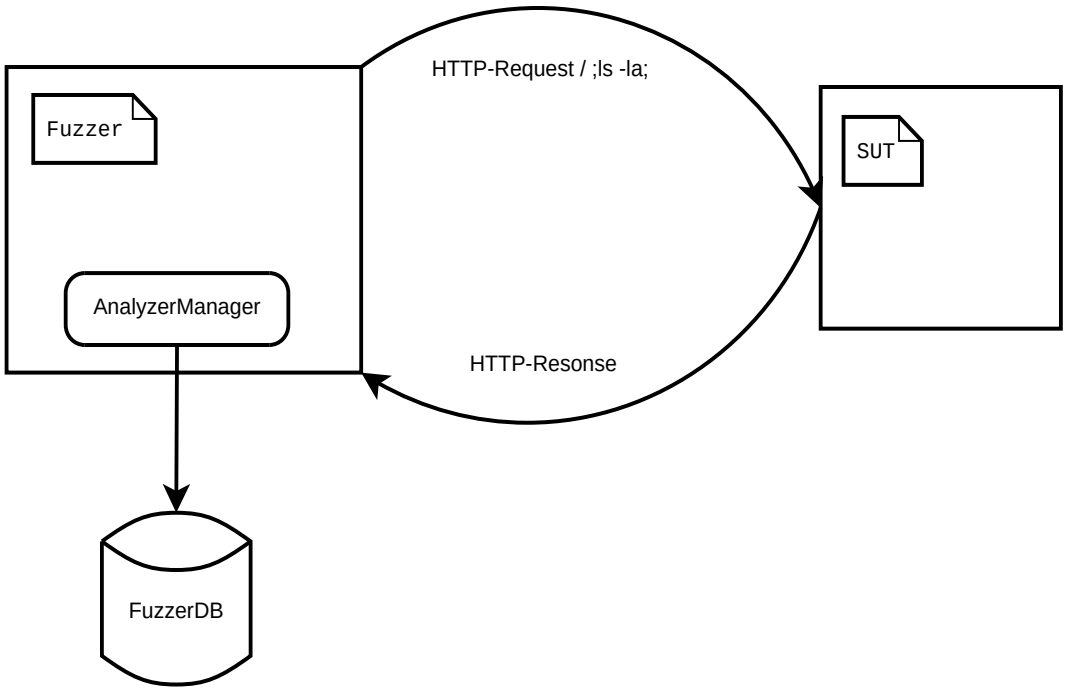

Abbildung 5.6: Schematischer Ablauf einer Command-Injection während eines Fuzzing-Testlaufs [vgl. [67]]

Im Listing 5.8 ist der in Abbildung 5.6 gezeigte HTTP-Request in Form eines Codebeispiels dargestellt.

```
1 GET /products/milch?q=product&n=;ls -la; HTTP/1.1
2 Host: 192.168.0.2:80
3 Accept: application/json
4 Connection: Keep-Alive
5 Content-Type: application/x-www-form-urlencoded
6 Accept: application/json, text/javascript, */*; q=0.01
7 X-Session: Cv1d5nPWgUZ8gjGvA=
8 X-Requested-With: XMLHttpRequest
```

Listing 5.8: Darstellung einer Command-Injection durch den Fuzzer in Form eines HTTP-Headers [vgl. [67]]

Der Request in Listing 5.8 enthält die kompromittierende Zeichenkette ”;ls -la;”, welche ähnlich wie in Listing 5.7 im Rahmen eines Testfalls am zu testenden System gestartet werden und dort möglicherweise einen unberechtigten Verzeichniswechseln auslösen könnte.

```
1 HTTP/1.1 400 Bad Request
2 Date: Mon, 07 Nov 2011 11:38:34 GMT
3 Connection: close
4 Content-Length: 0
```

Listing 5.9: Darstellung einer Antwortnachricht des SUT auf eine Command-Injection des Fuzzers in Form eines HTTP-Headers [vgl. [67]]

Als Antwort des SUT könnte der Fuzzer zum Beispiel die HTTP-Response wie aus Listing 5.9 erhalten haben. Diese gibt zu erkennen, dass offenbar ein fehlerhafter Request an das SUT geschickt worden ist.

Nach dieser beispielhaften Darstellung einiger möglicher Sicherheitsbedrohungen in Bezug auf Fuzzing wird im folgenden Kapitel zunächst schematisch erklärt, wie eine mögliche Analyse von Sicherheitstestdaten mittels Data-Mining erfolgen kann, welche zum Beispiel von Analyzern produziert worden sind. Die Anlyse und Evaluation von konkreten Sicherheitstestergebnissen wird im Kapitel 7 erläutert.

6 Erkennung von Sicherheitsfehlern durch Data-Mining

Im 4 Kapitel wurde die Repräsentation von Wissen behandelt. Zu den Darstellungsmöglichkeiten zählen Entscheidungsbäume, Cluster, Klassifikations- und Assoziationsregeln. In den nächsten Abschnitten werden Algorithmen zur Wissensverarbeitung vorgestellt, welche die Entdeckung von Auffälligkeiten in Daten behandeln.
Diese Algorithmen fanden bisher ihre erfolgreiche Anwendung in allgemeinen Data-Mining Bereichen (z. B. Klassifikation oder Clusteranalyse), aber auch im speziellen Bereich der Softwaresicherheit, um beispielsweise Softwarefehler basierend auf Softwaretestergebnissen zu finden. Eine Möglichkeit bietet das diskriminative Pattern Mining. Dabei werden Testergebnisse nach gültigen und fehlgeschlagenen Testfällen klassifiziert. Es wird versucht festzustellen, ob bestimmte Funktionsaufrufbäume der zu testenden Software öfter bei gültigen oder fehlgeschlagenen Testfällen aufgetreten sind. Von dieser Information ausgehend wird die Fehlerhaftigkeit der getesteten Software beurteilt. [vgl. [18]]
Es gibt auch noch andere Methoden, die versuchen eine Abschätzung und Prognose von Softwarefehlern zu ermöglichen, um beispielsweise den Korrekturaufwand [vgl. [64]] und die Komplexität [vgl. [56]] dieser Fehler zu prognostizieren. Zum einen wurde mit Klassifikationsalgorithmen (Bayes'sche Methode und Entscheidungsbäume) gearbeitet [vgl. [41, 40]], zum anderen wurden Assoziationsregeln verwendet und den zuvor erwähnten Klassifikationsverfahren gegenübergestellt. [vgl. [64]] In dieser Arbeit werden jedoch nur Algorithmen behandelt, welche mögliche Sicherheitsfehler in Datenmengen finden. Von der Abschätzung der Komplexität beziehungsweise Prognose des Korrekturaufwandes der Fehler wird abgesehen. Dieses Kapitel erläutert anhand von Beispielen, wie Entscheidungsbäume, Cluster oder Regeln aus Sicherheitstestergebnisdaten generiert werden können und das daraus gewonnene Wissen weiter analysiert sowie evaluiert werden kann.

6.1 Assoziationsregeln

Die einfachste Art Assoziationsregeln aus einer vorliegenden, aufbereiteten Datenmenge zu gewinnen, besteht in der Anwendung eines simplen Algorithmus. Die Qualität der generierten Assoziationsregeln lässt sich durch den bereits erläuterten Support respektive die zugehörige Konfidenz ermitteln.

6.1.1 Einfacher Algorithmus

Dieser Algorithmus kann als Brute-Force-Methode zur Regelermittlung bezeichnet werden. Die Anwendung des Verfahrens wird am folgenden Beispiel nach Gupta demonstriert. [vgl. [32]] Angenommen die zu analysierende Datenmenge besteht aus fünf Testfällen. Diese wurden im Rahmen eines Sicherheitstests durchgeführt und das Ergebnis (Fehlermeldungen) pro Testfall in einer Datenbank gespeichert. Pro Testfall können jeweils die Fehlermeldungen "Ausnahmefehler", "Eingabefehler", "Parsefehler" und "Sicherheitsfehler" enthalten sein. Diese sind von der zu testenden Applikation als Antwortnachrichten an das Testtool zurückgekommen und werden als Ausgangssituation in der Tabelle 6.1 dargestellt.

Testfall	Fehlermeldungen
1	Sicherheitsfehler, Ausnahmefehler, Eingabefehler
2	Eingabefehler, Sicherheitsfehler
3	Ausnahmefehler, Sicherheitsfehler, Parsefehler
4	Eingabefehler, Parsefehler
5	Eingabefehler, Sicherheitsfehler

Tabelle 6.1: Ausgangssituation bei der Generierung von Assoziationsregeln durch den einfachen Algorithmus [vgl. [32]]

Es wird versucht Assoziationen zwischen den aufgetretenen Fehlermeldungen zu finden. In Anlehnung an Gupta wird ein Support und eine Konfidenz von größer oder gleich 50% beziehungsweise 75% als Mindestmaß für gefundene Assoziationen veranschlagt. [vgl. [32]]
Im nächsten Schritt werden die Elemente aller Testfälle miteinander kombiniert. Dies ist in Tabelle 6.2 veranschaulicht. Es wird versucht die Häufigkeit der einzelnen Elementkombinationen zu ermitteln. Auf diesen Kombinationen aufbauend, wird anschließend deren Support und Konfidenz berechnet.

Kombination	Fehlermeldungen	Häufigkeit
1	Parsefehler	2
2	Eingabefehler	4
3	Ausnahmefehler	2
4	Sicherheitsfehler	4
5	(Parsefehler, Eingabefehler)	1
6	(Parsefehler, Ausnahmefehler)	1
7	(Parsefehler, Sicherheitsfehler)	1
8	(Eingabefehler, Ausnahmefehler)	1
9	(Eingabefehler, Sicherheitsfehler)	3
10	(Ausnahmefehler, Sicherheitsfehler)	2
11	(Parsefehler, Eingabefehler, Ausnahmefehler)	0
12	(Parsefehler, Eingabefehler, Sicherheitsfehler)	0
13	(Eingabefehler, Ausnahmefehler, Sicherheitsfehler)	1
14	(Ausnahmefehler, Sicherheitsfehler, Parsefehler)	1

Tabelle 6.2: Kombination von Testfallfehlermeldungen zur Generierung von Assoziationsregeln durch den einfachen Algorithmus [vgl. [32]]

Die berechnete Häufigkeit und der sich daraus ableitende Support geben an, dass nur die Kombination neun aus Tabelle 6.2 für eine weitere Analyse relevant ist. Der Support für die Kombination neun errechnet sich daher folgendermaßen [vgl. [32]]:

$$Support(Eingabefehler \cap Sicherheitsfehler) = 3/5$$

Alle anderen Kombinationen in Tabelle 6.2 weisen analog zu dieser Berechnung nicht den benötigten Mindestsupport auf und können vernachlässigt werden. Weiter wird ermittelt, ob die gefundene Regeln auch die benötigte Konfidenz von 75% aufweist. Sie wird wie folgt berechnet [vgl. [32]]:

$$Support(Eingabefehler) = 4/5$$

$$Konfidenz(Eingabefehler \rightarrow Sicherheitsfehler) = \frac{Support(Eingabefehler \cap Sicherheitsfehler)}{Support(Eingabefehler)} = 0.75$$

Nach der Kalkulation einer 75%-igen Konfidenz für $Eingabefehler \rightarrow Sicherheitsfehler$ (Kombination neun) kann festgestellt werden, dass einige Eingabefehler mitunter Sicherheitsfehler am zu testenden System verursacht haben. [vgl. [8, 32]]

6.1.2 Apriori-Algorithmus

Eine Weiterentwicklung des einfachen Algorithmus stellt das Apriori-Verfahren dar. Es verbessert das Laufzeitverhalten des einfachen Algorithmus. Dabei wird davon ausgegangen, dass es sich bei der Generierung von Assoziationsregeln um eine binäre Funktion handelt. Der zuvor definierte Support und die festgelegte Konfidenz geben dabei wiederum die zwei unteren Grenzen an. Eine Regel wird nur als relevant eingestuft, falls sie die beiden Grenzen wertmäßig überschreitet. Beim einfachen Verfahren ergibt sich dadurch ein exponentielles Laufzeitverhalten. Der Apriori-Algorithmus versucht daher mit so genannten Kandidatensets (Frequent Itemset) sowie mittels Pruning das Suchproblem zu optimieren. [vgl. [32]]
Im ersten Schritt wird versucht Kandidaten für ein Kandidatenset (Frequent Itemset) zu finden. Danach wird versucht Subsets dieser Kandidatensets zu finden, welche ebenso häufig auftreten wie das Superset. Ein Kandidatenset erfüllt daher stets den definierten Mindestsupport. Es gelten daher folgende Aussagen: Falls ein Kandidatenset den notwendigen Support aufweist, so besitzen alle (nicht leeren) Subsets auch den benötigten Support. [vgl. [36]] Andererseits sind alle Subsets eines Kandidatensets der Ebene k auch leer, falls das Kandidatenset auf Ebene $k-1$ leer ist. [vgl. [10, 37]] Für das Pruning bedeutet das folgendes: Es soll beispielsweise ein Kandidatenset der Ebene k aus einem Kandidatenset der Ebene $k-1$ erzeugt werden. Beim Pruning können daher alle Kandidatensets von Ebene k gelöscht werden, die nicht häufig genug in Ebene $k-1$ vorkommen.
Dieses Verfahren wird an einem Beispiel nach Gupta [vgl. [32]] illustriert. Es wird analog zu jenem aus Abschnitt 6.1.1 konstruiert mit dem Unterschied, dass die Anfangssituation aus vier Testfällen besteht, die jeweils Fehlermeldungen enthalten, wie "Ausnahmefehler", "Sicherheitsfehler", "Formatfehler", "Parsefehler" und "Eingabefehler".

Testfall	Fehlermeldungen
1	Ausnahmefehler, Formatfehler, Sicherheitsfehler
2	Ausnahmefehler, Sicherheitsfehler, Parsefehler
3	Formatfehler, Sicherheitsfehler, Eingabefehler
4	Formatfehler, Ausnahmefehler

Tabelle 6.3: Ausgangssituation zum Finden von Assoziationen zwischen Softwarefehlermeldungen mittels Apriori-Algorithmus [vgl. [32]]

Als L1 wird jene Menge bezeichnet, die die Häufigkeit der gefundenen Fehlermeldungen innerhalb der Testfälle widergibt. [vgl. [32]] Im konkreten Beispiel wären folgende Häufigkeiten [vgl. Tabelle 6.3]: "Ausnahmefehler" kommt dreimal vor, "Formatfehler" dreimal, "Sicherheitsfehler" dreimal, "Parsefehler" und "Eingabefehler" einmal. In Anlehnung an Gupta wird der Mindestsupport mit 50% bestimmt, die Mindestkonfidenz mit 80%. Im nächsten Schritt wird das bereits erwähnte Kandidatenset C2 erzeugt, welches in Tabelle 6.4 veranschaulicht wird. Dabei werden die Fehlermeldungen "Parsefehler" und "Eingabefehler" nicht berücksichtigt, da sie nicht den nötigen Support aufweisen.

Kombination	Fehlermeldungen	Häufigkeit
1	(Ausnahmefehler, Formatfehler)	2
2	(Ausnahmefehler, Sicherheitsfehler)	2
3	(Formatfehler, Sicherheitsfehler)	2

Tabelle 6.4: Generierung des Kandidatensets C2 zum Finden von Assoziationen zwischen Softwarefehlern mittels Apriori-Algorithmus [vgl. [32]]

Die drei in Tabelle 6.4 enthaltenen Fehlermeldungskombinationen besitzen alle einen Mindestsupport von 50%. Ausgehend von den Kandidaten kann ein weiteres Kandidatenset C3 generiert werden, nämlich mit dem Inhalt (Ausnahmefehler, Formatfehler, Sicherheitsfehler). Dies ist möglich, weil jeweils das erste Element der Kombination eins und zwei ident ist. [vgl. Tabelle 6.4, [32]] Für die erzeugte Kombination der drei Fehlermeldungen kann ein Support von 25% berechnet werden. Somit ist sie nicht relevant für eine weitere Analyse. Für die drei Kombinationen aus dem Kandidatenset C2 in Tabelle 6.4 kann folgende Konfidenz berechnet werden:

$Ausnahmefehler \rightarrow Formatfehler = 2/3$
$Formatfehler \rightarrow Ausnahmefehler = 2/3$
$Ausnahmefehler \rightarrow Sicherheitsfehler = 2/3$
$Sicherheitsfehler \rightarrow Ausnahmefehler = 2/3$
$Formatfehler \rightarrow Sicherheitsfehler = 2/3$
$Sicherheitsfehler \rightarrow Formatfehler = 2/3$

In diesem Beispiel erfüllt folglich keine Kombination (Assoziationsregel) die nötige Konfidenz. [vgl. [32, 10, 37]]

Die Verwendung des Apriori-Algorithmus kommt unter anderem auch für das Mining von Softwarerepositories in Frage. Dabei wurden Bug-Reports verschiedener Softwareprojekte hergenommen und die dort aufgetretenen Softwarefehler daraus manuell klassifiziert, im nächsten Schritt kam der Apriori-Algorithmus zum Einsatz. Es wurde folgende Analyse mitsamt Evaluation erstellt: Die Ursache von Softwarefehlern wurde vesucht zu erruieren, indem verschiedene Bedingungen ausgewertet wurden (z. B. Nullwerte, Grenzwerte). Außerdem wurde versucht herauszufinden, welche Auswirkungen verschiedene Softwarefehler hatten. Schließlich wurden Softwarefehler noch miteinander in Relation gesetzt. Es wurde auch untersucht in welchen funktionalen Modulen der Software Fehler aufgetreten sind. [vgl. [50, 49]] Assoziationsregeln eignen sich auch zum Text-Mining, hängen aber von der statistischen Verteilung der im Text enthaltenen Wörter ab. [vgl. [12]]

6.2 Klassifikation

Klassifikation bildet neben dem Finden von Assoziationsregeln ein weiteres Verfahren zur Extraktion von Wissen aus Datenmengen. Dabei können verschiedene Varianten dieses Verfahrens zum Einsatz kommen, wie: Klassifikation mittels Entscheidungsbäumen,
Bayes'sche Klassifikation, regelbasierte Klassifikation oder Support-Vektor-Maschinen (SVM). Im Rahmen dieser Arbeit wird näher auf die Verarbeitung von Entscheidungsbäumen und die Bayes'sche Methode eingegangen. [vgl. [35]] Dazu werden zum besseren Verständnis kurze Beispiele gebracht.
Die Klassifikation von Objekten wurde bereits erfolgreich im Bereich von Cyber-Security eingesetzt. Dabei wurden beispielsweise E-Mails mit der Bayes'schen Methode auf Würmer untersucht sowie eine Extraktion von böswilligem Code mittels Support-Vektor-Maschinen (SVM).

Außerdem wurde mit Hilfe von Klassifikation und Ausreißeranalyse versucht Botnets aufzuspüren; bei dieser Art von Analyse wurden keine statischen Daten, sondern kontinuierliche Datenströme verwendet. [vgl. [72]] Ein anderes Anwendungsgebiet von Klassifikation lässt sich in der Kryptographie finden, wo untersucht wurde, ob die Klassifikation von verschlüsselten Texten jener der entschlüsselten entspricht. [vgl. [45]]

6.2.1 Wahl der Klassifikationsattribute

Bei der Klassifikation gilt es jene Attribute zu finden, welche die Objekte am besten unterscheiden, damit ein Entscheidungsbaum aufgebaut werden kann. Das Maß für die Unterscheidungskraft von Attributen wird Informationsgehalt genannt. [vgl. [75]] Prinzipiell gibt es zwei Möglichkeiten um den Informationsgehalt von Attributen zu berechnen [vgl. [35]]:

- Informationstheorie
- Gini Index

Der Begriff der Informationstheorie basiert auf den Forschungen von Claude Shannon. Es wird dabei vom mittleren Informationsgehalt oder der Informationsdichte von Daten ausgegangen. Dabei spielt der Begriff der Unsicherheit eine große Rolle, denn wo Information vorliegt, gibt es auch Unsicherheit darüber. Liegt keine Unsicherheit vor, gibt es auch keine Information. Es kann beispielsweise ein Münzwurf betrachtet werden. Die Wahrscheinlichkeit, dass Kopf oder Zahl kommt, beträgt 50% und ist somit gleich hoch. Es kann aber nicht vorher gesagt werden, ob beim nächsten Wurf Kopf oder Zahl kommt. Somit besteht maximal Ungewissheit darüber und daher liegt Information vor. Wären auf der Münze zwei Köpfe, würde keine Information vorliegen. [vgl. [32]]
Für die Klassifikation von Objekten bedeutet das folgendes: Die Partition D besteht aus einer Menge von N Objekten mit Attributen $a_1, a_2, \ldots, a_n$, wobei jedes dieser Attribute einen unterschiedlichen Informationsgehalt besitzt. Für eine Klassifikation ist daher jenes Attribut zu bestimmen, welches den höchsten Informationsgehalt aufweist. Dadurch wird erreicht, dass die Unsicherheit über die Klassifikation der Objekte soweit als möglich minimiert wird. Dieses Attribut ist in der Lage die Objekte in der Datenmenge am eindeutigsten zu klassifizieren und verhindert, dass Objekte mehreren Klassen gleichzeitig zugewiesen werden können. [vgl. [35]]
Der Informationsgehalt einer Partition D berechnet sich wie folgt [vgl. [35]]:

$$Info(D) = -\sum_{i=1}^{m} p_i log_2(p_i)$$

$Info(D)$ entspricht dabei dem Informationsgehalt vor dem Teilen der Partition D. Um berechnen zu können nach welchem Attribut gesplittet werden kann, muss der Informationsgehalt des Split-Attributes vom ursprünglichen Gehalt der Partition D abgezogen werden. Dies geschieht folgendermaßen [vgl. [35]]:

$$Gehalt(A) = Info(D) - (\sum_{i=1}^{v} \frac{|D_i|}{|D|} * Info(D_i))$$

$\frac{|D_i|}{|D|}$ gibt dabei die Gewichtung der Partition D_i in Bezug auf die Ausprägungen des Split-Attributes an. Diese Formel ist für alle Attribute zu berechnen. Erst danach kann festgestellt werden, welches Attribut den größten Informationsgehalt besitzt; nach diesem wird in weiterer Folge partitioniert. Diese Schritte werden so oft wiederholt, bis alle Objekte eindeutig klassifiziert werden konnten. [vgl. [32, 35]]
Eine andere Variante zur Berechnung des Informationsgehalts von Attributen wird durch den

Gini Index ermöglicht. Dieser Index geht auf Corrado Gini zurück und gibt an, wie das Ungleichgewicht der Ressourcenverteilung innerhalb einer Population ist. In der Praxis wird der Gini Index verwendet, um beispielsweise die Verteilung der Gehälter in einer Volkswirtschaft zu berechnen. Der Index variiert jeweils zwischen null und eins, wobei null die größtmögliche Unsicherheit (die Gehälter sind alle gleichmäßig verteilt) und eins keine Unsicherheit (alle Gehälter gehören einer Person) in Bezug auf den Informationsgehalt darstellt. [vgl. [32]]
Werden die Objekte einer Partition D durch n Klassen klassifiziert, gibt p_i die Wahrscheinlichkeit an, dass ein Objekt in D zur Klasse C_i gehört. Der Gini Index für die Partition D errechnet sich wie folgt [vgl. [35]]:

$$Gini(D) = 1 - \sum_{i=1}^{n} p_i^2$$

Der Gini Index stellt prinzipiell einen binären Split für jedes Attribut dar. Das bedeutet, die Partition D wird immer pro Attribut in D_1 und D_2 aufgeteilt. Um zu berechnen wie hoch der Gini Index für ein Attribut A ist, kann folgende Formel verwendet werden [vgl. [35]]:

$$Gini_A(D) = \frac{|D_1|}{D} Gini(D_1) + \frac{|D_2|}{D} Gini(D_2)$$

Dieser Wert wird anschließend vom ursprünglichen Informationsgehalt $Gini(D)$ abgezogen. Danach kann bestimmt werden, welches Attribut den größtmöglichen Informationsgewinn geliefert hat. Dieses Attribut wird als Split-Attribut verwendet.

6.2.2 Klassifikation mit Entscheidungsbäumen

Ein Entscheidungsbaum besteht, wie schon zuvor in Abschnitt 4.5 erwähnt, aus einem Wurzelknoten, inneren Knoten und Blättern. Dabei wird immer für jedes zu klassifizierende Objekt von der Wurzel ausgegangen und versucht über innere Knoten zu einem Blatt zu kommen. Dieses Blatt weist dem Objekt seine Klasse zu.
Im nachfolgenden Beispiel wird eine Klassifikation mittels Entscheidungsbaum nach Gupta [vgl. [32]] und dem Ansatz der Informationstheorie aus Abschnitt 6.2.1 demonstriert. Ausgehend von der Tabelle 6.5 wird illustriert, wie beispielsweise bereits durch einen Fuzzer bewertete (F-Rating) Testfälle mittels Entscheidungsbaum klassifiziert werden können.

Testfall	Eingabefehler	Laufzeitfehler	Parsefehler	F-Rating	Klasse
1	ja	nein	ja	A	Sicherheitsfehler
2	nein	ja	ja	B	kein Sicherheitsfehler
3	ja	ja	nein	A	mögl. Sicherheitsfehler
4	nein	nein	ja	A	Sicherheitsfehler
5	ja	nein	nein	C	mögl. Sicherheitsfehler
6	nein	nein	nein	C	kein Sicherheitsfehler
7	ja	ja	ja	A	kein Sicherheitsfehler

Tabelle 6.5: Ausgangssituation bei der Klassifikation von Sicherheitstestergebnissen mittels Entscheidungsbaum [vgl. [32]]

Dabei dient die Klassifikation dazu, dass entschieden werden kann, ob es sich bei bestimmten Testfällen tatsächlich um das Auftreten von Sicherheitsfehlern auf Grund der Fuzzerbewertungen gehandelt hat oder nicht. Dazu wurden die Daten aus der Tabelle 6.5 zusätzlich manuell klassifiziert, was in der Spalte "Klasse" ersichtlich ist. Diese manuelle Klassifikation sagt dabei aus, dass der erste Testfall vom Fuzzer als Sicherheitsfehler eingestuft wurde ("F-Rating = A") und dieser tatsächlich einen Sicherheitsfehler am SUT verursacht hat ("Klasse = Sicherheitsfehler").
Der Algorithmus zur Abarbeitung der Klassifikation kann wie folgt beschrieben werden [vgl. [75]]:

- Wähle ein Attribut aus der Menge der vorhandenen Attribute aus.
- Lege dieses Attribut als Wurzelknoten fest.
- Teile die verbleibenden Attribute je nach Ausprägungen des Wurzelknotenattributes in weitere Teilbäume auf.
- Wiederhole diesen Ablauf für die verbleibenden Attribute pro Teilbaum solange bis alle Objekte restlos klassifiziert werden konnten.

Anhand von Tabelle 6.5 wird die zuvor erwähnte Beispielklassifikation durchgeführt. [vgl. [32]] In dieser Tabelle existieren sieben Testfälle mit drei Klassen. Dabei treten die drei Klassen in den nachfolgenden Häufigkeiten auf:

Sicherheitsfehler = 2
möglicher Sicherheitsfehler = 2
kein Sicherheitsfehler = 3

Die nachfolgend verwendeten Formeln leiten sich aus dem Abschnitt 6.2.1 ab. Der Informationsgehalt aus den Daten (Klassen) der Tabelle 6.5 beträgt daher:

$$Info(D) = -(2/7)\log(2/7) - (2/7)\log(2/7) - (3/7)\log(3/7) \approx 0.47$$

In weiterer Folge ist das Split-Attribut zu bestimmen, welches die gesamte Partition D am eindeutigsten klassifiziert. Für das Attribut "Eingabefehler" aus der Tabelle 6.5 ergibt sich dadurch folgende Berechnung.
Beim Vergleich "Eingabefehler = ja" treten die Klassen in den Häufigkeiten auf: "Sicherheitsfehler = 1", "mögl. Sicherheitsfehler = 2" und "kein Sicherheitsfehler = 1". Bei "Eingabefehler = nein" gibt es die Klassenhäufigkeiten: "Sicherheitsfehler = 1", "mögl. Sicherheitsfehler = 0" und "kein Sicherheitsfehler = 2".

$$\begin{aligned} Info_{Eingabefehler} = {} & (4/7) \times (-(2/4)\log(2/4) - (1/4)\log(1/4) - (1/4)\log(1/4)) \\ & + (3/7) \times (-(1/3)\log(1/3) - (2/3)\log(2/3)) \approx 0.38 \end{aligned}$$

Der errechnete Split der Partitionen nach dem Attribut "Eingabefehler", würde einen Informationsgewinn von

$$Gehalt(Eingabefehler) = 0.47 - 0.38 = 0.09$$

bringen. Für die anderen Attribute ergibt die analoge Berechnung einen Informationsgehalt von ≈ 0.09 für den "Laufzeitfehler", ≈ 0.18 für den "Parsefehler" und ≈ 0.26 für das "F-Rating". Das Split-Attribut mit dem meisten Informationsgehalt ist daher "F-Rating". Der erste Knoten im Entscheidungsbaum ist das "F-Rating" mit drei Kindknoten und deren Ausprägungen "A", "B", "C". Die weitere Anwendung des Algorithmus ergibt, dass das nächste Split-Attribut

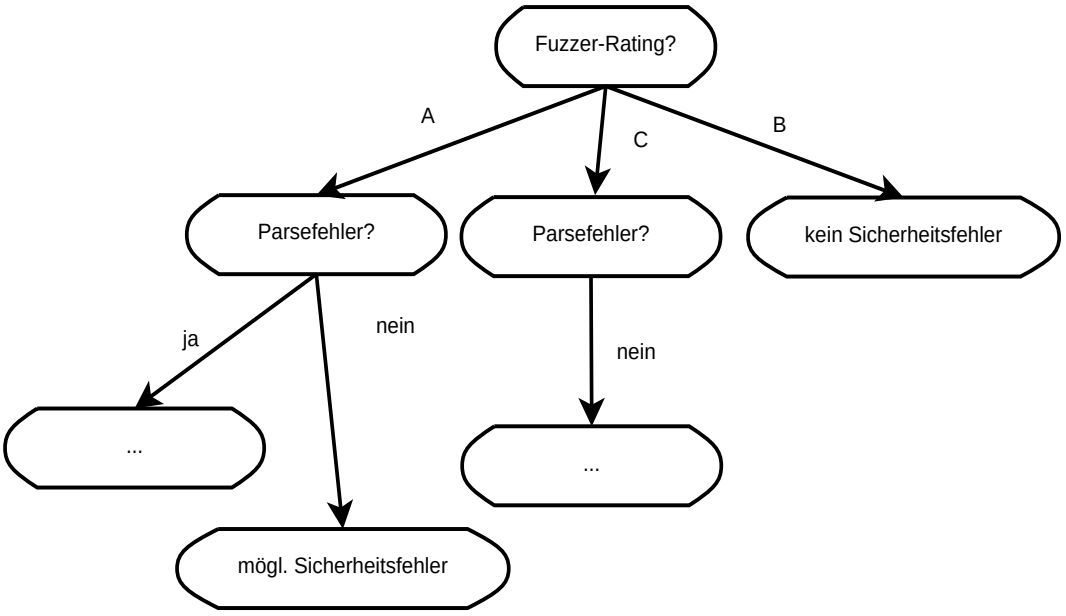

Abbildung 6.1: Graphische Darstellung des Entscheidungsbaums nach der Klassifikation von Softwarefehlern [vgl. [32, 35, 10]]

”Parsefehler” ist. Der Entscheidungsbaum hätte danach das in Abbildung 6.1 dargestellte Erscheinungsbild. Die nächsten Abarbeitungsschritte des Algorithmus erfolgen analog zum ersten Schritt und werden nicht weiter verfolgt.
Klassifikationsalgorithmen können auch in Verbindung mit Log-Dateien angewendet werden, um Anomalien im Bereich Sicherheitsmonitoring zu finden. Dabei wurden die Inhalte aus Log-Dateien von unterschiedlichen Netzwerktools (z. B. Firewall, Virenscanner usw.) analysiert. [vgl. [52]]

6.2.3 Klassifikation nach der Bayes’schen Methode

Ausgehend von der Tabelle 6.5 auf Seite 52 wird die Klassifikationsmethode nach Bayes vorgestellt. [vgl. [32, 10, 35]] Hierbei kommt das Bayes’sche Theorem zum Einsatz. Im nachfolgenden Beispiel wird ebenso ein Testfall als möglicher, tatsächlicher oder kein Sicherheitsfehler klassifiziert.
X ist ein Objekt, welches klassifiziert werden soll. Dieses Objekt wird durch n Attribute charakterisiert. H bezeichnet die Hypothese, dass X durch die Klasse C klassifiziert werden kann. Es wird die Wahrscheinlichkeit $P(H|X)$ ermittelt, mit welcher ein Objekt X zur Klasse C_i angehört, bedingt durch seine Attribute $a_1, a_2, \ldots, a_n$. Diese bedingte Wahrscheinlichkeit kann mit dem Bayes’schen Theorem berechnet werden. Angenommen es exisiteren k Klassen. Für jede der Klassen ist folgende Berechnung auszuführen [vgl. [32, 10, 35]]:

$$P(C_i|X) = \frac{P(X|C_i)P(C_i)}{P(X)}$$

$P(C_i)$ beschreibt dabei die Apriori-Wahrscheinlichkeiten der k Klassen. Dabei werden für eine Klasse alle Objekte dieser Klasse gezählt und danach durch die Gesamtanzahl aller Objekte dividiert. $P(X|C_i)$ kann durch die bedingte Unabhängigkeit ausgedrückt werden. Es wird vereinfacht angenommen, dass alle Attribute bedingt unabhängig sind.

Dies wird durch folgende Formel dargestellt [vgl. [32, 10, 35]]:

$$P(X|C_i) = \prod_{m=1}^{n} P(x_m|C_i) = P(x_1|C_i) \times P(x_2|C_i) \times \cdots \times P(x_m|C_i)$$

Ein Objekt gehört zur Klasse C_i, falls gilt $P(C_i|X) \leq P(C_j|X)$ für alle $1 \leq j \leq k, j \neq i$. [vgl. [35]] Unter Verwendung der Tabelle 6.5 wird folgender, neuer Testfall klassifiziert:

$$\mathbf{X} = (Eingabefehler = nein, Laufzeitfehler = ja,$$
$$Parsefehler = nein, Fuzzer - Rating = A)$$

$P(C_i)$ für die Klassen "Sicherheitsfehler", "möglicher Sicherheitsfehler" und "kein Sicherheitsfehler" ergibt sich jeweils zu:

$P(Sicherheitsfehler) = 2/7 \approx 0.29$
$P(m.\ Sicherheitsfehler) = 2/7 \approx 0.29$
$P(kein\ Sicherheitsfehler) = 3/7 \approx 0.43$

Es kann $P(X|C_i)$ berechnet werden:

$P(Eingabefehler = nein|Klasse = Sicherheitsfehler) = 1/2 = 0.5$
$P(Eingabefehler = nein|Klasse = m.\ Sicherheitsfehler) = 0/2 = 0$
$P(Eingabefehler = nein|Klasse = kein\ Sicherheitsfehler) = 2/3 \approx 0.6$

$P(Laufzeitfehler = ja|Klasse = Sicherheitsfehler) = 0/2 = 0$
$P(Laufzeitfehler = ja|Klasse = m.\ Sicherheitsfehler) = 1/2 = 0.5$
$P(Laufzeitfehler = ja|Klasse = kein\ Sicherheitsfehler) = 2/3 \approx 0.6$

$P(Parsefehler = nein|Klasse = Sicherheitsfehler) = 0/2 = 0$
$P(Parsefehler = nein|Klasse = m.\ Sicherheitsfehler) = 2/2 = 1$
$P(Parsefehler = nein|Klasse = kein\ Sicherheitsfehler) = 1/3 \approx 0.3$

$P(Fuzzer - Rating = A|Klasse = Sicherheitsfehler) = 2/2 = 1$
$P(Fuzzer - Rating = A|Klasse = m.\ Sicherheitsfehler) = 1/2 = 0.5$
$P(Fuzzer - Rating = A|Klasse = kein\ Sicherheitsfehler) = 1/3 \approx 0.3$

$$P(X|Sicherheitsfehler) =$$
$$P(Eingabefehler = nein|Klasse = Sicherheitsfehler) \times$$
$$P(Laufzeitfehler = ja|Klasse = Sicherheitsfehler) \times$$
$$P(Parsefehler = nein|Klasse = Sicherheitsfehler) \times$$
$$P(Fuzzer - Rating = A|Klasse = Sicherheitsfehler) = 0$$

Analog dazu erfolgt die Berechnung für $P(X|m.\ Sicherheitsfehler)$ = 0 und für $P(X|kein\ Sicherheitsfehler) \approx 0.032$. Da die ersten beiden Klassen wegfallen, ergibt sich für den neuen Testfall X die Klassifikation "kein Sicherheitsfehler" mit einer Wahrscheinlichkeit von $P(X|kein\ Sicherheitsfehler) = 0.43 \times 0.032 \approx 0.014$.
Die Bayes'sche Klassifikationsmethode hat im Vergleich zu anderen Klassifikationsalgorithmen eine hohe Genauigkeit und Performance, auch wenn sie auf große Datenmengen angewendet wird. [vgl. [32, 35]]
Sie fand ihre Anwendung bereits im Bereich von mobilen Netzwerken, bei welchen der Datentransfer auf Anomalien (z. B. Softwareangriffe) überprüft wurde. Dabei wurden mobile Agenten eingesetzt, welche miteinander kommunizierten und dadurch herausfinden konnten, ob ein Angreifer versucht hat einen Netzwerkknoten zu attackieren. Diese Methode liefert gegenüber anderen Verfahren eine geringere Rate an falsch positiven Überprüfungsergebnissen. [vgl. [23]]

6.3 Clusteranalyse

Ein weiteres Verfahren zum Finden von Mustern in Datenmengen stellt die Clusteranalyse dar. Hierbei handelt es um die Partitionierung einer Grunddatenmenge in einzelne Gruppen. Einerseits sind die Eigenschaften der einzelnen Elemente innerhalb einer Gruppe ähnlich zueinander. Andererseits unterscheiden sich Elemente von verschiedenen Gruppen deutlich. Mit Hilfe dieser Segmentierung wird verifiziert, ob sich die vorliegenden Daten in natürliche Klassen einteilen lassen. Durch diese Kategorisierung können weitere Schlüsse über den Zusammenhang der Daten gezogen werden, beispielsweise zur Klärung der Frage, ob ein Element in Cluster A oder B gehört - ob es daher ähnlicher den Elementen in A oder jenen in B ist. Hierzu bedarf es eines Vergleichsmaßes, welches auch Distanz genannt wird. [vgl. [37, 65]]

6.3.1 Distanzmaß

Die Distanz zwischen zu kategorisierenden Elementen gibt an, wie ähnlich diese Elemente zueinander sind. Für die Berechnung dieser Distanz werden jene Attribute verwendet, welche die Elemente bestmöglich charakterisieren. Falls erforderlich, können die Daten standardisiert werden (mit einem Mittel von null und einer Standardabweichung von eins). [vgl. [32]]

Außerdem besitzt das Distanzmaß folgende Eigenschaften [vgl. [32, 65]]:

- Die Distanz ist immer positiv und die Distanz eines Punktes zu sich selbst ist immer Null.
- Die Distanz eines Punktes x zu einem Punkt y ist nicht größer, als die Summe der Distanz von Punkt x zum Punkt z und die Distanz von Punkt z zu Punkt y.
- Die Distanz von Punkt x zu Punkt y ist immer die gleiche wie von y zu x.

Die euklidische Distanz ist ein verbreitetes Maß zur Berechnung der Distanz zwischen Objekten. Für die Berechnung können beispielsweise deren Attribute X und Y herangezogen werden. Große Werte können jedoch gegenüber kleineren das Berechnungsergebnis verzerren. Eine gute Skalierung der Attribute erweist sich hier als sinnvoll. Mit der nachfolgenden Formel kann die euklidische Distanz berechnet werden [vgl. [32]]:

$$D(X,Y) = \sqrt{\sum_i (x_i - y_i)^2}$$

Die Summe der Quadrate minimiert dabei den Einfluss von großen Attributwertschwankungen auf das Ergebnis. Das euklidische Distanzmaß wird meist bei nicht standardisierten Daten verwendet. [vgl. [32, 10]]
Die Manhattan-Distanz oder City-Block-Distanz ist ähnlich wie die euklidische Distanz aufgebaut. Der einzige Unterschied besteht darin, dass bei der Berechnung keine Quadrate, sondern nur absolute Attributwerte herangezogen werden. Die folgende Formel kann zur Berechnung der Manhattan-Distanz benützt werden [vgl. [32, 10]]:

$$D(X,Y) = \sum_i (|x_i - y_i|)$$

6.3.2 Arten der Clusteranalyse

Die soeben erwähnten Distanzmaße finden bei Clusteranalyseverfahren ihren Einsatz. Neben einigen existierenden Clusterbildungsverfahren werden zwei davon erläutert, um den Zusammenhang zur nachfolgenden K-Means-Methode herzustellen.

Partitionsverfahren segmentieren eine Datenmenge, bestehend aus n Elementen, in k disjunkte Teilmengen, so dass jede Teilmenge in sich so homogen als möglich ist. Die Partitionierung erfolgt auf Grund einer Funktion. Sie berechnet die bereits erwähnte Distanz zwischen den Elementen. Im konkreten Fall wird die Distanz von den Elementen zu einem jeweiligen Clustermittelwert berechnet. Die Elemente werden jenem Cluster zugewiesen, zu welchem sie die geringste Distanz besitzen.[vgl. [35, 32]]
Hierarchische Verfahren gehen im Gegensatz zum partitionierenden Verfahren von einem Supercluster aus und unterteilen diesen in mehrere Subcluster. Dies wird auch als top-down Ansatz bezeichnet. Eine andere Möglichkeit besteht darin, dass sich anfangs jedes Element in einem eigenen Cluster befindet. Es wird sukzessive versucht ähnliche Cluster zu einem größeren Clustern zu verschmelzen. Diese Vorgehensweise wird auch als bottom-up bezeichnet.[vgl. [32, 65]]

6.3.3 K-Means-Methode

Im Rahmen dieser Arbeit wird die K-Means-Methode genauer behandelt. Von den übrigen partitionierenden und hierarchischen Verfahren wird abgesehen, weil sie den Umfang der Arbeit sprengen würden.
Der K-Means-Algorithmus hat seinen Namen auf Grund der K Cluster, welche generiert werden. Das Wort "Mean" bezieht sich hierbei auf den Mittelwert, durch welchen jeder erzeugte Cluster repräsentiert wird. Es wird dabei davon ausgegangen, dass zu jedem Berechnungszeitpunkt der Mittelwert jedes Clusters bekannt ist. Falls das beim ersten Berechnungsschritt nicht der Fall ist, werden die ersten n Elemente als Startmittelwerte angenommen. In jedem Schritt wird daher für die übrigen Elemente berechnet, wie weit diese zu den Startmittelwerten entfernt sind. Die Elemente werden danach jenem Startmittelwert beziehungsweise dem dadurch repräsentierten Cluster zuordnet, der sich am nächsten befindet. Nach diesem Zuordnungsschritt werden die Mittelwerte für die nächste Iteration erneut berechnet. Danach erfolgt wieder die Zuweisung der Elemente zu Clustern. Dieser iterative Prozess wird solange wiederholt, bis sich die Elemente innerhalb der Cluster nicht mehr verändern. Bei einem Suchproblem dieser Art wird bei jeder Iteration versucht ein lokales Optimum zu erzeugen. Dieses Optimum muss global gesehen nicht immer optimal sein, kann jedoch effizient kalkuliert werden. Zur Berechnung der Distanz wird üblicherweise die euklidische Distanz verwendet. Es kann aber auch die Manhattan-Distanz benützt werden. Diese ist allerdings empfindlicher gegenüber Ausreißern in der Datenmenge. [vgl. [32, 75, 65]]

Der K-Means-Ansatz wird an einem Beispiel nach Bramer schematisch vorgezeigt. [vgl. [10]]. Es handelt sich dabei um Daten, welche im Rahmen eines Fuzzing-Testlaufs als Testergebnisse ermittelt wurden. Die Tabelle 6.6 enthält dabei als Ausgangssituation die Resultate dreier Analyzer. Der erste Analyzer (ResponseTimeAnalyzer) hat die Länge der Antwortzeiten pro Testfall gemessen, der zweite (ResponseAnalyzer) die Länge der Antwortnachrichten und der dritte die Anzahl der gefundenen Fehlermeldungen (FileReaderAnalyzer). Diese gewonnenen Informationen werden mittels Clusteranalyse analysiert.
Bei der Berechnung auftretende Kommazahlen werden jeweils zur ihren nächstgelegeneren Grenzen ab- beziehungsweise aufgerundet. In diesem Beispiel wird der Einfachheit halber die Manhattan-Distanz verwendet.

Testfall	ResponseTimeAnalyzer	ResponseAnalyzer	FileReaderAnalyzer
1	1	4	3
2	2	5	2
3	1	1	3
4	2	4	4
5	1	3	2
6	4	1	3

Tabelle 6.6: Ausgangssituation bei der Clusteranalyse von Sicherheitstestergebnissen nach der K-Means-Methode [vgl. [10]]

Die ersten drei Testfälle werden als Startmittelwerte angenommen und sind in Tabelle 6.7 dargestellt.

Testfall	ResponseTimeAnalyzer	ResponseAnalyzer	FileReaderAnalyzer	Cluster
1	1	4	3	C1
2	2	5	2	C1
3	1	1	3	C1

Tabelle 6.7: Initiale Mittelwerte bei der Clusteranalyse von Sicherheitstestergebnissen nach der K-Means-Methode [vgl. [10]]

Danach wird die erste Iteration gestartet. Es wird jeweils die Entfernung aller Elemente zu den drei Mittelwerten berechnet. In der letzten Tabellenspalte wird die vorläufige Clusterzuordnung dargestellt. Das wird in Tabelle 6.8 veranschaulicht.

	Entfernung			
Testfall	von C1	von C2	von C3	Cluster
1	0	3	3	C1
2	3	0	7	C2
3	3	7	0	C3
4	2	3	5	C1
5	1	3	3	C1
6	6	7	3	C3

Tabelle 6.8: Clustereinteilung von Sicherheitstestergebnissen mittels K-Means-Methode nach der ersten Iteration [vgl. [10]]

In nächsten Schritt erfolgt die erneute Berechnung der Mittelwerte, was durch Tabelle 6.9 abgebildet ist.

Testfall	ResponseTimeAnalyzer	ResponseAnalyzer	FileReaderAnalyzer	Cluster
1	1	3	3,5	C1
2	3	0	7	C2
3	4,5	7	1,5	C3

Tabelle 6.9: Mittelwerte bei der Clusteranalyse von Sicherheitstestergebnissen mittels K-Means-Methode nach der ersten Iteration [vgl. [10]]

Danach wird analog zur ersten die zweite Iteration begonnen; dies wird in Tabelle 6.10 dargestellt.

	Entfernung			
Testfall	von C1	von C2	von C3	Cluster
1	0,5	10	10	C1
2	8,5	0	14	C2
3	9,5	14	3	C3
4	2,5	6	10	C1
5	0,5	9	3	C1
6	9,5	14	3	C3

Tabelle 6.10: Clustereinteilung von Sicherheitstestergebnissen mittels K-Means-Methode nach der zweiten Iteration [vgl. [10]]

Die finale Clusterung ergibt sich nach der zweiten Iteration hiermit in Tabelle 6.11.

Cluster	Testfall
C1	1, 4, 5
C2	2
C3	3, 6

Tabelle 6.11: Finale Clusterung von Sicherheitstestergebnissen mittels K-Means-Methode [vgl. [10]]

Wie aus der zweiten Iteration ersichtlich ist, haben sich die Elemente der Cluster von Runde eins zu Runde zwei nicht verändert. Daher kann die Clusterung aus Runde zwei als endgültig betrachtet werden. In diesem Beispiel wurde die Skalierung entsprechend gewählt. Dies muss bei Verfahren zur Clusteranalyse aber nicht immer der Fall sein. Es kann vorkommen, dass einige Attribute wesentlich größeren Einfluss auf den Clustermechanismus besitzen, als andere. Für eine effektive Clusterung ist es daher von Bedeutung, dass die verwendeten Attribute relativ gleich gewichtet sind. [vgl. [32, 75]] Attribute können zu diesem Zweck skaliert und somit deren Gewichtung verändert werden.

Mögliche Skalierungsmechanismen sind [vgl. [32, 75]]:

- Division jedes Attributes durch dessen Mittelwert
- Division jedes Attributes durch die Differenz des größten und kleinsten Attributwertes
- Transformation des Attributes durch Standardisierung (Subtraktion des Attributmittels von jedem Attributwert und Division der Differenz durch die Standardabweichung)

Bei der Clusteranalyse stellt sich anfangs immer die Frage der Wahl der optimalen Startwerte. Es kann nicht immer davon ausgegangen werden, dass sie anfangs passend zur Verfügung stehen. Startwerte können zum Beispiel randomisiert erzeugt werden. Danach wird mit dem Clustern gestartet. Nach Beendigung der Clusteranalyse werden erneut zufällige Startwerte generiert. Dieses Verfahren kann sich über mehrere Iterationen erstrecken, bis die finalen Cluster gefunden wurden.
Es ist auch möglich Startwerte zu wählen, die eine große Distanz zueinander ausweisen. Cluster, die nahe beieinander liegen, können beim iterativen Vorgehen auch miteinander verschmolzen werden.

Ein weiterer Ansatz schlägt vor, dass aus einer Datenmenge Trainingsdaten extrahiert werden. Mit Hilfe dieser Trainingsdaten und zufällig erzeugten Startwerten wird anschließend versucht Startwerte für die gesamten Daten zu finden. [vgl. [32, 10]]

Im Rahmen von Datensicherheit wurde die Clusterung von Daten bereits erfolgreich eingesetzt. Es konnten einerseits damit, ähnlich wie beim Finden von Assoziationen oder bei der Klassifikation, Log-Daten von Netzwerktools analysiert werden, um beispielsweise DoS-Angriffe aufzuspüren. [vgl. [48]]
Bei einem anderen Versuch konnte festgestellt werden, welche Art von Maleware Netzwerktools überhaupt mitgeloggt haben. Diese Information konnte mittels Assoziationsregeln extrahiert werden. Außerdem wurde noch erforscht, in welche Cluster sich einzelne Maleware einordnen hat lassen. Die so gewonnenen Informationen wurden im Rahmen von Security-Information-And-Event-Management (SIEM) weiter verarbeitet. [vgl. [27]]

6.4 Text-Mining

Eine etwas abgewandelte Form von Data-Mining stellt das so genannte Text-Mining dar. Im Rahmen dieser Arbeit werden die bereits vorgestellten Data-Mining-Verfahren in Verbindung mit Text-Mining gebracht. Beim klassischen Data-Mining wird versucht Objekte zu klassifizieren, Cluster zu bilden oder Relationen zwischen Objekten zu finden. Beim Klassifizieren beziehungsweise bei der Clusteranalyse wird von einer eindeutigen Objektzuweisung ausgegangen. Dabei kann beispielsweise ein Objekt exklusiv nur einem Cluster zugeordnet sein. Die Klassifikation von Objekten verläuft analog dazu; ein Objekt kann immer nur einer Klasse angehören. Der Unterschied zum Text-Mining besteht darin, dass eine Textsorte auch mehreren Kategorien angehören kann. Die Exklusivität ist nicht immer gewährleistet, in manchen Fällen gar nicht erwünscht. Beim Text-Mining werden analog zum Data-Mining n Kategorien unterschieden, wie zum Beispiel: Wirtschaft, Musik, Bildung, Geschichte und so weiter. Es ist daher möglich, dass sich ein Textteil in mehrere dieser Kategorien einordnen lässt.
Klassifikationen dieser Art sind meist sehr zeit- und rechenintensiv. Diese Form von Text-Mining wird auch als Text-Data-Mining (TDM) bezeichnet und beruht auf einem induktiven Lernmechanismus. Eine weitere Möglichkeit von Text-Mining bietet der Ansatz des
Text-Knowledge-Mining (TKM). Hierbei wird Wissen mittels Abduktion und Deduktion gefolgert. Textelemente werden nicht nur auf Grund ihres Vorkommens innerhalb eines Text analysiert, es wird auch die sich dahinter befindliche Semantik der Textelement evaluiert. Beispielsweise könnten beim TDM die Regel $Eingabefehler \rightarrow Systemabsturz$ gemeinsam mit der Regel $Systemabsturz \rightarrow Eingabefehler$ gefunden werden. Es ist daher trivial, dass ein Eingabefehler zu einem Systemabsturz führen kann und nicht umgekehrt. Bei TKM kann auch Hintergrundwissen über die zu analysierenden Daten berücksichtigt werden. [vgl. [10, 66]]

6.4.1 Textdarstellung im Rahmen von Text-Mining

Data-Mining basiert standardmäßig auf einer fixen Nummer von Objektattributen. Text-Mining konzentriert sich hingegen auf Textmerkmale. Der Text kann daher als eine Aneinanderreihung von Wörtern (Bag-Of-Words) bezeichnet werden. [vgl. [19]] Die Anzahl dieser Merkmale kann mitunter sehr groß und variabel sein. Einige dieser Merkmale tauchen in einem konkreten Text möglicherweise sehr selten auf, andere wiederum sind sehr häufig vertreten. Diese Eigenschaft von Text wird auch als Noise bezeichnet.
Um diesem Umstand beizukommen, wird der Fließtext in eine geeignetere Form gebracht. Der Text wird in einzelne Wörter zerlegt und zum Beispiel in eine Tabelle geschrieben. Nach diesem Transformationsschritt kann zum Beispiel berechnet werden wie oft ein gewisses Wort im Text vorgekommen ist. Es kann aber auch die Anzahl von speziellen Wortkombinationen oder

Phrasen gezählt werden. Eine weitere Variante wäre das Auftreten von bestimmten Buchstabenkombinationen (N-Grammen) zu ermitteln. Dabei werden beim simplen TDM Konstrukte, wie Absätze, Satzzeichen und die Semantik des Textes ignoriert. Wie bereits erwähnt, wird die Semantik und eventuelles Hintergrundwissen beim TKM schon verwendet. Auf diese Weise lässt sich Text wortbasiert und einfach darstellen. [vgl. [66]]
In dieser Arbeit wird der Ansatz des induktiven Text-Mining (TDM) näher behandelt. Es bestehen drei mögliche Ansätze, um Text-Mining durchzuführen: Wörterbücher, regelbasiert, oder klassifikationsbasiert. [vgl. [19]] Der Wörterbuchansatz wird nachfolgend näher erläutert.
In Wörterbüchern, oder auch Dictionaries genannt, werden alle Wörter, die im zu analysierenden Text vorkommen, zumindest einmal erfasst. Dies kann in einem lokalen oder globalen Wörterbuch erfolgen.
Im globalen Wörterbuch befinden sich prinzipiell N Kategorien, zu jeder dieser Kategorien können X Wörter erfasst werden. Beim Text-Mining werden in weiterer Folge beliebige Textelemente mit den Wörtern der einzelnen Kategorien verglichen. Auf diese Art und Weise können neue Wörter gelernt und ins Wörterbuch übernommen werden. [vgl. [19]]
Bei lokalen Wörterbüchern allerdings wird für jede der N Kategorien ein eigenes Wörterbuch erzeugt. Es enthält jeweils alle Wörter, welche einer bestimmten Kategorie angehörig sind. Zu beachten ist, dass die Generierung von N lokalen Wörterbüchern mitunter zeitaufwändig ist. Lokale Wörterbücher benötigen jedoch im Vergleich zu einem globalen Wörterbuch weniger Speicherplatz und eignen sich daher performancemäßig besser, als globale. [vgl. [10]]

6.4.2 Selektionsmechanismen für Wörter aus Texten

Nachdem der unstrukturierte Text in eine anschaulichere Form (z. B. in eine Tabelle) gebracht wurde, gilt es jene Wörter daraus zu extrahieren, die für weitere Entscheidungen von Relevanz sind. Bevor jedoch interessante Wörter aus dem Text extrahiert werden können, bedarf es einer Bereinigung und Reduktion der vorhandenen Wörter, da nicht alle Wörter für ein Text-Mining von Bedeutung sind. Insbesondere können das Endungen von Wörtern beziehungsweise Binde-, oder Füllwörter sein. [vgl. [19]]

6.4.3 Stoppwort-Entfernung und Vereinheitlichung der Schreibweise

Text-Mining geht vom Ansatz der wortbasierten Darstellung von Text aus. Viele dieser Wörter sind für das maschinelle Lernen jedoch nicht von Bedeutung. Sie können mitunter das Laufzeitverhalten von Algorithmen beeinflussen. Daher sollten diese Wörter möglichst minimiert werden. Ebenso gilt es die Schreibweise von Wörtern zu vereinheitlichen und die Wörter entweder alle groß oder klein zu schreiben. [vgl. [33]]
Stoppwörter bezeichnen jene Wörter, die für eine Klassifikation nutzlos sind und vor dem Wörterbuchaufbau aus dem Text entfernt werden können. Es existiert keine endgültige Liste von Stoppwörtern. Jede Sprache besitzt welche, die Anzahl kann von Sprache zu Sprache variieren. Im englischen Sprachraum können beispielsweise „a“, „an“, „is“, „the“ oder „you“ als Stoppwörter betrachtet werden. Die gewählten Stoppwörter sollten allerdings das Text-Mining nicht negativ beeinträchtigen, zum Beispiel durch eine Fehlklassifikation von Text, weil die Liste der Stoppwörter fälschlicherweise auch wichtige Klassifikationswörter enthält. [vgl. [10]]

6.4.4 Wortstammreduktion

Nachdem die Stoppwörter aus dem Text extrahiert wurden, kann als weitere Phase vor der Erzeugung des Wörterbuches eine Wortstammreduktion durchgeführt werden. Sie reduziert ähnliche Wörter auf ihren gemeinsamen Wortstamm und zielt auf syntaktisch ähnliche Wörter ab, wie: Verben, Plural von Nomen, Wortvariationen usw. Zum Beispiel besitzen die Wörter ”fehlen”, ”Fehler”, ”Eingabefehler”, ”Sicherheitsfehler”, ”Pufferüberlauffehler” alle denselben

Wortstamm ”fehl”. Enthält ein Text ein paar dieser Wörter, kann davon ausgegangen werden, dass er sich eventuell in den Bereich Softwaresicherheit (z. B. Fehlerbehandlung) einordnen lässt. Wörter wie ”Fehler”, ”Sicherheitsfehler”, ”fehlen” können daher als identisch betrachtet werden. [vgl. [33, 19]] Für Wortstammreduktionen sind viele Algorithmen bekannt; einer davon ist der Stemmer nach Martin Porter [vgl. [53, 62, 63]]

6.4.5 Filterung mittels Vector-Space-Model

Text kann in Form von lokalen oder globalen Wörterbüchern dargestellt werden. In diesen Wörterbüchern können einzelne Wörter sowie ganze Phrasen oder Passagen enthalten sein. Sie werden auch Terme oder Attribute genannt und im Wörterbuch mittels $t_1, t_2, \ldots, t_N$ beschrieben. Der N-dimensionale Vektor $(X_{i1}, X_{i2}, \ldots, X_{iN})$ bezeichnet daher die geordnete Reihenfolge von N Termen des i-ten Textes. Diese Schreibweise wird auch als das Vector-Space-Model oder VSM bezeichnet. Jedes Attribut beschreibt, wie oft ein Term in einem bestimmten Text vorgekommen ist. Generell bedeutet das: X_{ij} gibt die Häufigkeit des j-ten Terms im i-ten Text an. Diese Anzahl der Terme wird auch als Termhäufigkeit (Term Frequency) bezeichnet. Sie errechnet sich entweder durch das Aufsummieren der einzelnen Terme oder kann binär mittels null oder eins dargestellt werden. Dabei gibt null das Vorhandensein und eins das Nichtvorhandensein eines Terms im Text an. [vgl. [19]] Ein Nachteil der Termhäufigkeit ist, dass sie allen Termen die gleiche Wichtigkeit zuordnet, auch jenen die den Text vielleicht nicht optimal charakterisieren. Daher kann in diesem Fall die inverse Dokumenthäufigkeit (Inverse Document Frequency) verwendet werden. Mit diesem Mechanismus können charakteristische Wörter eine höhere Gewichtung als andere im Gesamttext zugewiesen bekommen. [vgl. [53]]

6.4.6 Vektornormalisierung und -normierung

Bevor N-dimensionale Vektoren verwendet werden können, ist es sinnvoll sie zu normalisieren. Diese Vorgehensweise ermöglicht unterschiedliche Attribute gleich zu gewichten, um eine Verzerrung des Text-Mining- Ergebnisses zu vermindern. Zum Beispiel können Log-Dateien welche mittels Fuzzing erzeugt wurden durch Text-Mining auf Ähnlichkeit überprüft werden [vgl. [10]].

Testlauf	Log-Dateien	SQL-Injection	Command-Injection	Pufferüberlauf
1	201	5	30	1
2	76	3	12	57

Tabelle 6.12: Ausgangssituation bei der Normierung von Vektoren [vgl. [10]]

Angenommen es wurden beispielsweise bei einer Erhebung von Sicherheitslücken Log-Dateien durchsucht. Dazu wurden im Vorfeld zwei Sicherheitstestläufe mit einem Fuzzer durchgeführt. Im ersten Lauf ergaben sich 201 Log-Dateien, im zweiten 76. In den Log-Dateien wurden Fehlermeldungen zu unterschiedlichen Sicherheitsbedrohungen (z. B. Pufferüberläufte, SQL-Injection usw.) gefunden. Werden diese Tabelleneinträge genauer betrachtet, erhalten die Ausprägungen des Attributes Log-Datei durch ihren Zahlenwert im Gegensatz zu den Werten der anderen Attribute eine starke Gewichtung. Es ist daher nicht zweckmäßig eine Klassifikation nach nur diesem Attribut durchzuführen. Alle anderen Attribute sind dabei ebenso entscheidend, daher ist eine Normalisierung der Attributwerte zu empfehlen. [vgl. [10]]

Dazu kann die euklidische Distanzformel verwendet werden [vgl. [10]]:

$$\sqrt{w_1(a_1 - b_1)^2 + w_2(a_2 - b_2)^2 + \cdots + w_n(a_n - b_n)^2}$$

Mit Hilfe dieser Formel können zwei Objekte A und B mit deren Attributen $a_1, a_2, \ldots, a_n$ beziehungsweise $b_1, b_2, \ldots, b_n$ (und einer möglichen Gewichtung derselbigen) miteinander verglichen werden. a_i und b_i setzen sich bei nummerischen Attributen jeweils aus der folgenden Formel zusammen [vgl. [75]]:

$$a_i = \frac{v_i \ - min\ v_i}{max\ v_i - min\ v_i}$$

v_i beschreibt dabei den Wert des aktuellen Attributes und min beziehungsweise max geben jeweils den minimalen beziehungsweise maximalen Wert des Attributes bezogen auf die gesamte zu vergleichende Objektmenge an. Bei nominalen Attributen kann beim Vergleich jeweils eins für Gleichheit oder null für Ungleichheit gewählt und in die Formel eingesetzt werden. [vgl. [75]] Beim Vergleich von Vektoren kann dieses Prinzip ähnlich angewendet werden. Ein normierter Vektor hat einen Wert zwischen null und eins - inklusive dieser Grenzen. Es sei ein weiteres Beispiel zur Verdeutlichung angeführt. [vgl. [10, 53]]

Ein Text beinhaltet sieben Terme und wird durch den Vektor $(0, 0, 2, 4, 0, 3, 1)$ dargestellt. Dabei treten der erste, zweite und fünfte Term jeweils null Mal auf; der dritte zweimal, der vierte viermal, der sechste dreimal und der siebte einmal. Ein neuer Text wird erstellt, welcher sich aus dem Produkt von zweimal dem alten Text ergibt. Somit verdoppeln sich die Werte des neuen Vektors und der zugehörige Vektor lautet daher $(0, 0, 4, 8, 0, 6, 2)$. Falls die Attribute im Text verzehnfacht würden, hätte der Vektor die Form $(0, 0, 20, 40, 0, 30, 10)$. Dieser Vektor hat mit der Darstellung des ursprünglichen nichts mehr gemeinsam. Ein solcher Umstand ist nicht zweckmäßig, da die beiden Texte als "gleich" zu klassifizieren sind.
Die Normierung des Vektors löst dieses Problem. Dazu wird die Länge jedes Vektors berechnet indem die Quadratwurzel aus der Summe der quadrierten Vektorelemente gezogen wird. Danach werden die einzelnen Attributwerte durch die Vektorlänge dividiert. Die Länge des resultierenden Vektors ist daher immer eins (Einheitsvektor).

Für die drei angeführten Vektoren bedeutet das folgendes [vgl. [10]]:

$$(0, 0, 2, 4, 0, 2, 1) = \sqrt{2^2 + 4^2 + 2^2 + 1^2} = 5$$

Die Länge ist fünf, daher ist der normalisierte Vektor (0, 0, 2/5, 4/5, 0, 2/5, 1/5).

$$(0, 0, 4, 8, 0, 4, 2) = \sqrt{4^2 + 8^2 + 4^2 + 2^2} = 10$$

Analog dazu hat dieser Vektor die Länge zehn und der normalisierte Vektor lautet (0, 0, 4/10, 8/10, 0, 4/10, 2/10).

$$(0, 0, 20, 40, 0, 40, 10) = \sqrt{20^2 + 40^2 + 40^2 + 10^2} = 50$$

Die Berechnung des letzten, normalisierten Vektor erfolgt analog zu den anderen, bereits kalkulierten Beispielvektoren.

6.4.7 Textvergleich mit Hilfe des Vector-Space-Models

Das Vector-Space-Model (VSM) kann zum Vergleich von Texten beim Data-Mining eingesetzt werden. Um dies zu ermöglichen, werden Vektorlängen miteinander verglichen. Die sich dadurch ergebende Distanz von Vektoren zueinander gibt an, wie ähnlich sich die Vektoren respektive die Texte sind, welche durch die Vektoren beschrieben werden. Identische Vektoren besitzen daher eine Distanz von null und komplett unterschiedliche von eins und alle anderen befinden sich innerhalb dieser Grenzen. Um einen Textvergleich anstellen zu können, wird das

Kreuzprodukt der zu vergleichenden Vektoren gebildet und von eins subtrahiert.

”Das Kreuzprodukt wird definiert als die Summe der Produkte der korrespondierenden Wertpaare zweier Einheitsvektoren derselben Dimension.” [Manning, 2008, [53]]

Zum Beispiel werden die zwei Vektoren $(0, 3, 2, 4)$ und $(1, 2, 2, 0)$ normalisiert und ergeben $(0, 0.6, 0.4, 0.7)$ und $(0.3, 0.6, 0.6, 0)$. Das Kreuzprodukt beträgt daher $0 \times 0.3 + 0.6 \times 0.6 + 0.4 \times 0.6 + 0.7 \times 0 = 0.6$. Danach wird dieses Ergebnis von eins subtrahiert und ergibt die Distanz der beiden Vektoren zueinander. Im konkreten Beispiel wäre das $1 - 0.6 = 0.4$ und hieße, dass sich die Vektoren eher ähnlich sind. [vgl. [10]]

7 Anwendung von Data-Mining am Beispiel eines Fuzzing-Werkzeugs

In diesem Kapitel wird die Anwendung von Data-Mining-Methoden auf ein Sicherheitstestergebnis behandelt. Beim Fuzzing werden die Testergebnisse der durchgeführten Sicherheitstests von den jeweiligen Analyzern des Fuzzers bewertet, je nach Wahrscheinlichkeit, ob bestimmte Analyzer einen Sicherheitsfehler vermutet haben oder nicht. Dazu wird jeder Analyzer bezüglich seiner Aussagen gewichtet. Das bedeutet, dass die Bewertungen der einzelnen Analyzer pro Testfall unterschiedlich starke Aussagekraft besitzen. Der Fuzzer berechnet auf Grund der Gewichtung jedes Analyzers und dessen Bewertung pro Testfall, ob dieser möglicherweise einen Sicherheitsfehler oder Systemabsturz ausgelöst hat. Die Problematik dabei ist, dass der Fuzzer in seinen Entscheidungen zu falsch positiven beziehungsweise falsch negativen Testergebnisbewertungen kommen kann. Solche Bewertungen verzerren das Endergebnis und müssen daher analysiert und gegebenenfalls optimiert werden.
Ziel ist die Analyse und Optimierung von Sicherheitstestergebnissen, welche mit Hilfe des im Kapitel 5 vorgestellten Fuzzing-Tools erzeugt worden sind. In diesem Kapitel wird daher zuerst ein Überlick über die durchgeführten Schritte gegeben, danach werden die Schritte einzeln näher erläutert. Zur schematischen Darstellung werden jeweils Codebeispiele, Tabellen und Abbildungen angeführt. Abschließend wird eine Evaluation der analysierten Ergebnisse durchgeführt.

7.1 Vorgehen der Analyse

Das Ziel dieser Arbeit ist ein Proof-Of-Concept. Es wurden dabei einige Algorithmen auf die vorhandene Datenbasis angewendet, um zu zeigen, dass aus den daraus gewonnenen Ergebnissen noch weitere Infomation zur Präzisierung des Sicherheitstestergebnisses abgeleitet werden kann.
Bei der Analyse wurden in Summe rund 8000 Testfälle betrachtet. Die Gewichtung der einzelnen Analyzer war relativ ähnlich zueinander. Zur Reduktion der falsch positiven Ergebnisse wurden zuerst alle Testfallergebnisse genau analysiert, wo der Fuzzer eine Vermutung auf Sicherheitsfehler geäußert hat. Anschließend wurden noch jene Testfälle stichprobenartig analysiert, wo der Fuzzer keinen Verdacht auf Sicherheitsfehler geäußert hat.
Im ersten Schritt wurde eine Datenbereinigung durchgeführt, welche in Anlehnung an Abschnitt 4.4 ausgeführt wurde. Als Nächstes wurde der Apriori-Algorithmus aus Abschnitt 6.1.2 auf die Daten angewendet. Der einfache Algorithmus aus Abschnitt 6.1.1 wurde vernachlässigt, weil er die Ausgangsbasis für den verbesserten Apriori-Algorithmus darstellt.
In weiterer Folge wurde eine Klassifikation der Daten durchgeführt, welche sich an den in den Abschnitten 6.2.2 und 6.2.3 erläuterten Methoden orientiert hat. Eine Clusteranalyse mittels K-Means aus Abschnitt 6.3.3 sowie Text-Mining aus Abschnitt 6.4 wurden für den Proof-Of-Concept dieser Arbeit nicht angewendet, sollten aber Bestandteil für weiterführende Arbeiten in diesem Bereich sein.

7.2 Bereiningung und Transformation der Daten

Bevor mit dem Analyseprozess begonnen wurde, wurden die Daten mittels ETL [vgl. [68]] in eine besser verarbeitbare Form gebracht. Beim Bereinigungsvorgang innerhalb der Datenbank

wurden die Analyzertabellen hinsichtlich ihrer Relevanz in Bezug auf die folgende Datenanalyse überprüft. Redundante Inhalte und Inkonsistenzen wurden so weit als möglich entfernt. Es wurden die bereinigten Tabellen alle mit dem Präfix ”ETL” versehen, um sie in der Datenbank kenntlich zu machen.

Testfall	Attacke	Request	Generator	Resultat
1	<$	SOAP-Request	security.fuzzer.RandomCharGenerator	A
2	%00/	SOAP-Request	security.fuzzer.RandomCharGenerator	C
3	<	SOAP-Request	security.fuzzer.RandomCharGenerator	B
4	..%255c	SOAP-Request	security.fuzzer.RandomDirectoryGenerator	A
5	12414	SOAP-Request	security.fuzzer.RandomNumberGenerator	B

Tabelle 7.1: Schematische Darstellung der Tabelle TESTCASE_VARIATION vor der Durchführung von ETL [vgl. [32]]

Die Tabelle 7.1 stellt schematisch die Ausgangssituation der Datenbanktabelle TESTCASE_VARIATION dar. Diese wurde mittels ETL in die Tabelle 7.2 transformiert. Die Spalte ”Testfall” spiegelt die Testfallnummer wider. Die Spalte ”Attacke” beschreibt das erzeugte, kompromittierende Angriffsmuster des jeweiligen Testfalls. Die Spalte ”Request” hat die Attacke in Form eines SOAP-Requests [vgl. Listing 5.5] beinhaltet, welches der Fuzzer zur zu testenden Applikation schickt. Die Spalte ”Generator” bezeichnet die Softwarekomponente, welche die Attacke erstellt hat. Und zuletzt wird noch die Gesamtbewertung des Fuzzers pro Testfall durch die Spalte ”Resultat” beschrieben.

Testfall	Attacke	Generator	Resultat
1	<$	RandomCharGenerator	A
2	%00/	RandomCharGenerator	C
3	<	RandomCharGenerator	B
4	..%255c	RandomDirectoryGenerator	A
5	312414	RandomNumberGenerator	B

Tabelle 7.2: Schematische Darstellung der Tabelle TESTCASE_VARIATION nach der Durchführung von ETL [vgl. [32]]

Im Transformationsschritt von Tabelle 7.1 nach Tabelle 7.2 wurde die Spalte ”Request” eliminiert. Darin waren die Testfälle des Fuzzers in Form von SOAP-Requests enthalten, welche jedoch für die nachfolgende Analyse nicht verwendet werden konnten. Die Spalte ”Generator” wurde mittels Stringfunktionen auf eine sinnvolle Länge gekürzt (z. B. Entfernung von ”security.fuzzer”). Alle anderen Spalten wurden unverändert übernommen.

7.3 Analyse des Datenbestandes mittels Data-Mining

Bei der Analyse der Ergebnisse wurden die in Abschnitt 7.1 erwähnten Methoden angewendet.

Assoziationsregeln

Der zuerst erprobte Ansatz zur Generierung von Assoziationsregeln wird schematisch in Abbildung 7.1 veranschaulicht und nachfolgend beschrieben sowie mit Hilfe von Tabellen illustriert.

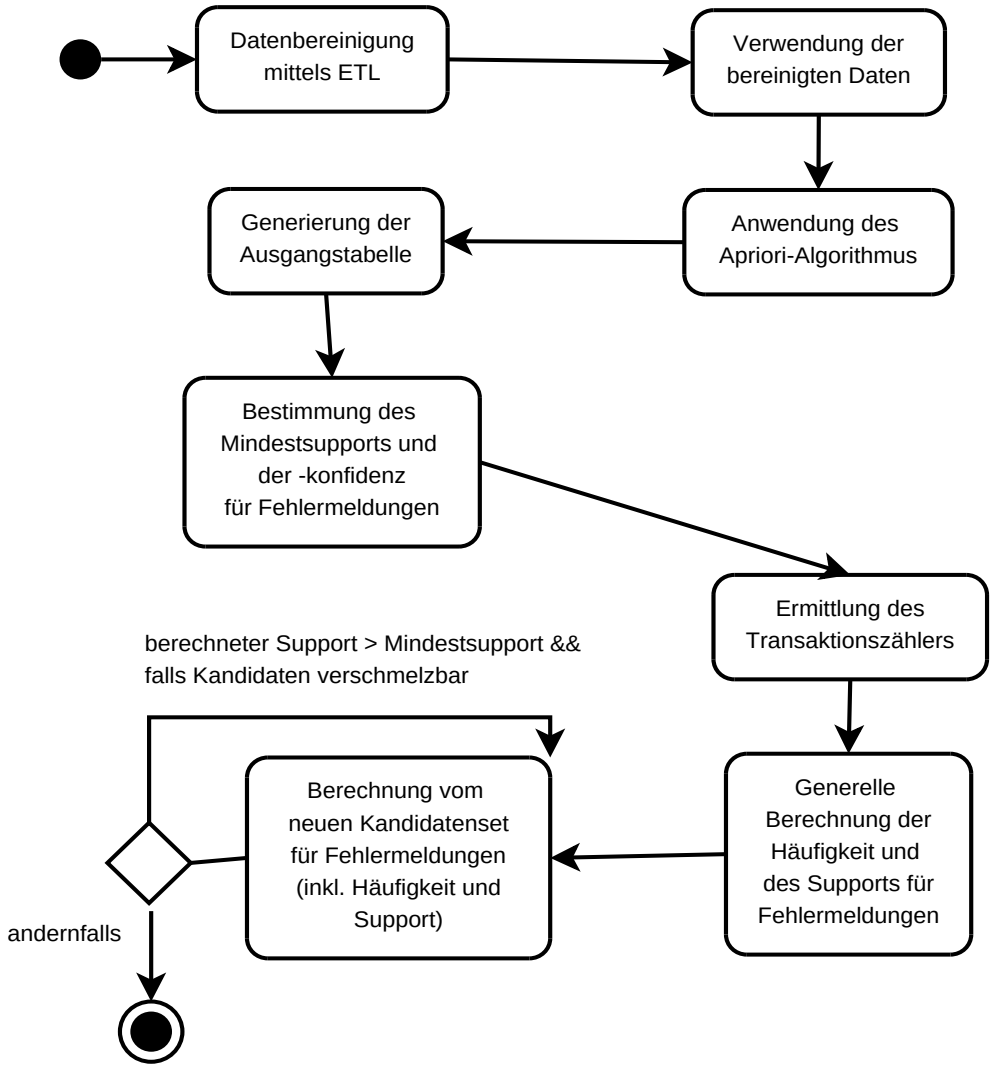

Abbildung 7.1: Schematische Darstellung des Analyseprozesses mittels Apriori-Algorithmus [vgl. [32]]

Es wurde versucht Assoziationsregeln aus den Daten abzuleiten, um jene Fehlermeldungen der zu testenden Applikation zu finden, die im Rahmen eines Testlaufs häufig und gemeinsam mit anderen Fehlermeldungen aufgetreten sind. Dazu wurde der Ansatz des Apriori-Algorithmus aus Abschnitt 6.1.2 gewählt. Als Beispiel wird hier der FileReaderAnalyzer verwendet, der bei jedem Testfall loggt, welche Fehlermeldungen die zu testende Applikation ausgegeben hat.

Eine schematische Darstellung der Ausgangstabelle zur Durchführung des Apriori-Algorithmus wird in Tabelle 7.3 gegeben.

Testfall	Fehlermeldungen
1	apache; exception; java.lang; numberformatexception; userexception
2	apache; exception; userexception
3	apache; exception; java.lang; numberformatexception
4	apache; exception; userexception; extranetexception
5	apache; exception; userexception; extranetexception
6	apache; exception
7	extranetexception; exception; userexception

Tabelle 7.3: Schematische Vorbereitung der Ausgangstabelle zur Durchführung des Apriori-Algorithmus [vgl. [32]]

Es wurde bei den Berechnungen ein Support von 60% und eine Konfidenz von 80% als Mindestkriterien für gefundene Assoziationsregeln angenommen. Diese Zahlen orientieren sich an jenen von Gupta. [vgl. [32]] Die Gesamtanzahl der durchgeführten Testfälle wurde ebenso ermittelt und wird in dieser Arbeit für den Apriori-Algorithmus als Transaktionszähler bezeichnet. [vgl. [32]] Dieser Zähler wurde zusätzlich in einer Tabelle gespeichert. Eine weitere Tabelle wurde angelegt, in welcher die Häufigkeit und der Support pro gefundener Fehlermeldungsart generell berechnet wurde. Sie wird schematisch durch Tabelle 7.4 dargestellt.

Fehlermeldung	Häufigkeit	Support
numberformatexception	167	0.7
userexception	145	0.5
exception	192	0.8
apache	136	0.43
java.lang	24	0.12

Tabelle 7.4: Schematische Ermittlung von aufgetretenen Fehlermeldungen gruppiert nach deren Häufigkeit zur Durchführung des Apriori-Algorithmus [vgl. [32]]

Im nächsten Schritt wurden die Fehlermeldungskombinationen für das Kandidatenset C2 ermittelt und deren Support berechnet. [vgl. Abschnitt 6.4] Dies wird in der Tabelle 7.5 schematisch veranschaulicht.

Kombination	Fehlermeldungen	Häufigkeit	Support
1	numberformatexception;userexception	143	0.74
2	numberformatexception;exception	143	0.74
3	numberformatexception;apache	192	0.8
4	numberformatexception;java.lang	136	0.43
5	userexception;exception	24	0.12
6	userexception;apache	34	0.18
7	userexception;java.lang	143	0.74

Tabelle 7.5: Schematische Ermittlung von Fehlermeldungskombinationen gruppiert nach deren Häufigkeit zur Durchführung des Apriori-Algorithmus [vgl. [32]]

Diese Schritte wären nach dem Algorithmus aus Abschnitt 6.1.2 zu wiederholen gewesen, bis hin zur Generierung des letztmöglichen Kandidatensets. [vgl. Abschnitt 6.4] Die weitere Anwendung des Apriori-Algorithmus wurde jedoch abgebrochen, weil dadurch keine sinnvollen Erkenntnisse gewonnen werden konnten, mit folgender Begründung: Zur Überprüfung, ob durch Testfälle Sicherheitsfehler verursacht worden sind, macht es keinen Sinn zu sagen, das zum Beispiel die Assoziationsregel ”numberformatexception → userexception” mit einem Support von 50% und einer Konfidenz von 75% in den Testergebnisdaten gefunden werden konnte. Das bedeutet nur, dass in 50% der Testfallergebnisse ”numberformatexception” und ”userexception” vorkommen und immer wenn ”numberformatexception” vorkommt, besteht zu 75% die Chance, dass auch ”userexception” vorkommt. [vgl. 4.5.3] Mit dieser Erkenntnis kann aber trotzdem nicht festgestellt werden, ob zum Beispiel durch Testfälle, mit diesen Fehlermeldungen im Ergebnis, ein Sicherheitsfehler am SUT ausgelöst worden ist oder nicht. Im Gegenteil, der Apriori-Algorithmus erachtet zum Beispiel die Assoziationsregel ”userexception → securityexception” für weitere Entscheidungen als unbedeutend, falls diese nicht den festgelegten Mindestsupport respektive die nötige Mindestkonfidenz aufweist. Wäre beispielsweise die Fehlermeldung ”securityexception” in Testergebnissen aufgetreten, wäre dies schon zu einem früheren Zeitpunkt aufgefallen, zum Beispiel beim Ausführen der Abfrage zur Erzeugung der Tabelle 7.4.

Im nächsten Versuch Auffälligkeiten in den Daten zu finden, wurde jeder Analyzer auf seine spezifischen Eigenschaften untersucht. Als Erstes wurden der FileReaderAnalyzer und dessen Ergebnisse analysiert. Dieser Prozess wird exemplarisch in der Abbildung 7.2 dargestellt und nachfolgend mit Tabellen verdeutlicht.

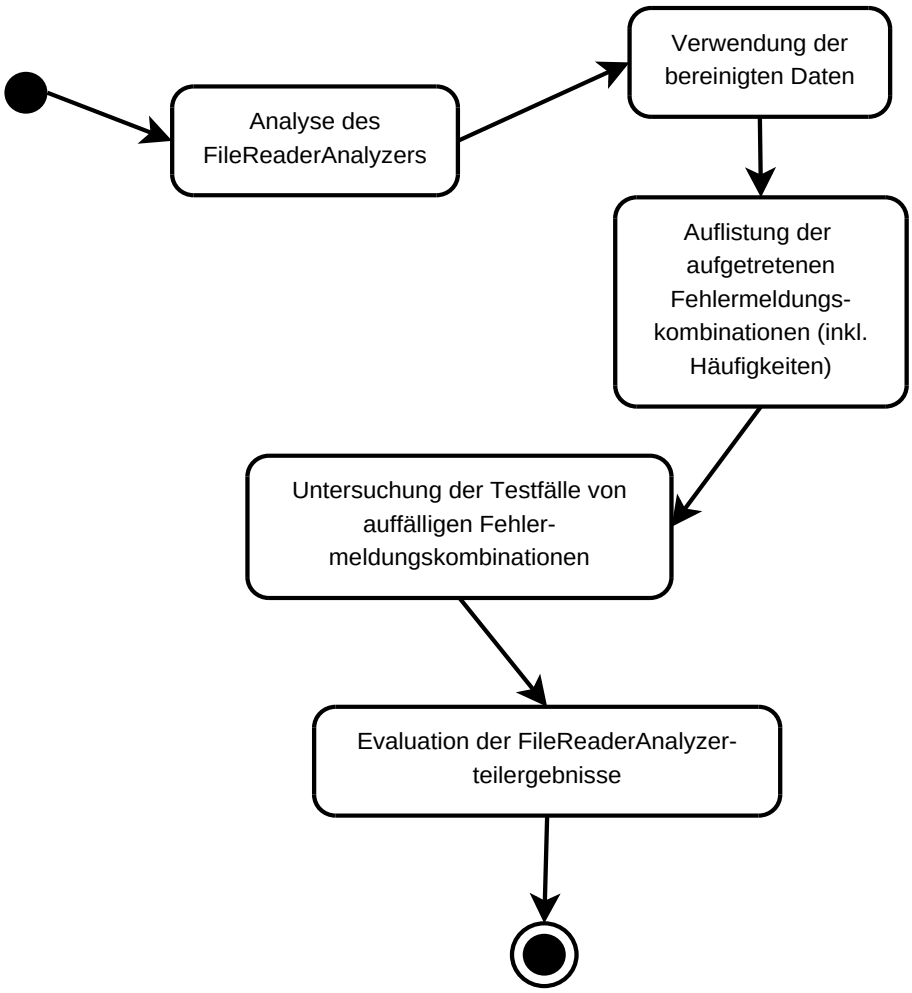

Abbildung 7.2: Schematische Darstellung des Analyseprozesses von Ergebnissen des FileReaderAnalyzers

Beim FileReaderAnalyzer wurden die aufgetretenen Fehlermeldungskombinationen nach ihrer Häufigkeit aufgelistet. Dies wird in Tabelle 7.6 exemplarisch veranschaulicht.

Kombination	Fehlermeldungen	Häufigkeit
1	apache; exception; java.lang; faultcode; userexception	136
2	apache; exception; userexception; extranetexception	32
3	apache; exception; userexception; saxparseexception	21
4	apache; exception; userexception	2
5	apache; exception; invalid; userexception; saxparseexception	1

Tabelle 7.6: Schematische Ermittlung der Häufigkeit pro aufgetretener Fehlermeldungen beim FileReaderAnalyzer zur Überprüfung von Auffälligkeiten

Pro im Ergebnis enhaltener Fehlermeldungskombination wurde analysiert, wie oft diese in bestimmten Testfällen vorgekommen sind und ob es sich dabei um mögliche Sicherheitsfehler gehandelt hat oder nicht. Zum Beispiel wurde ermittelt, welche Attacken (Testfälle) konkret die Fehlermeldungskombination ”apache; exception; invalid; userexception; saxparseexception” hervorgerufen haben. Diese ist auf Grund von Tabelle 7.6 nur bei einem Testfall vorgekommen. Dieser Testfall wurde anschließend als Ganzes betrachtet. Dazu wurde anschließend die Plausi-

bilität aller Analyzerbewertungen gesondert überprüft. Solche Ausreißer im Bezug auf die Häufigkeit sind im Rahmen der Analyse des Öfteren aufgetreten. Es konnte jedoch evaluiert werden, dass eine derartige Kombination von Fehlermeldungen mit ziemlicher Wahrscheinlichkeit keinen Sicherheitsfehler hervorgerufen haben und vom Fuzzer als falsch positiv eingestuft worden sind. Bei Fehlermeldungskombinationen, welche häufiger aufgetreten sind [vgl. Kombination eins in Tabelle 7.6] konnte herausgefunden werden, dass der Fuzzer diese auch als potenzielle Sicherheitsfehler eingestuft hat. Sie waren aber ebenso falsch positiv.
Bei den anderen Testfallergebnissen hat der FileReaderAnalyzer kaum oder nur geringe Differenzen zum normalen Verhalten des SUT entdeckt und diese somit nicht als Sicherheitsfehler eingestuft. Sie wurden trotzdem der Vollständigkeit halber ebenso überprüft und die Entscheidungen des Fuzzers schienen hier gerechtfertigt.
Als nächster Analyzer wurde der ResponseAnalyzer untersucht, welcher die Antwortnachrichten von der zu testenden Applikation auf deren Länge beziehungsweise deren Differenz zum normalen Verhalten des SUT überprüft.

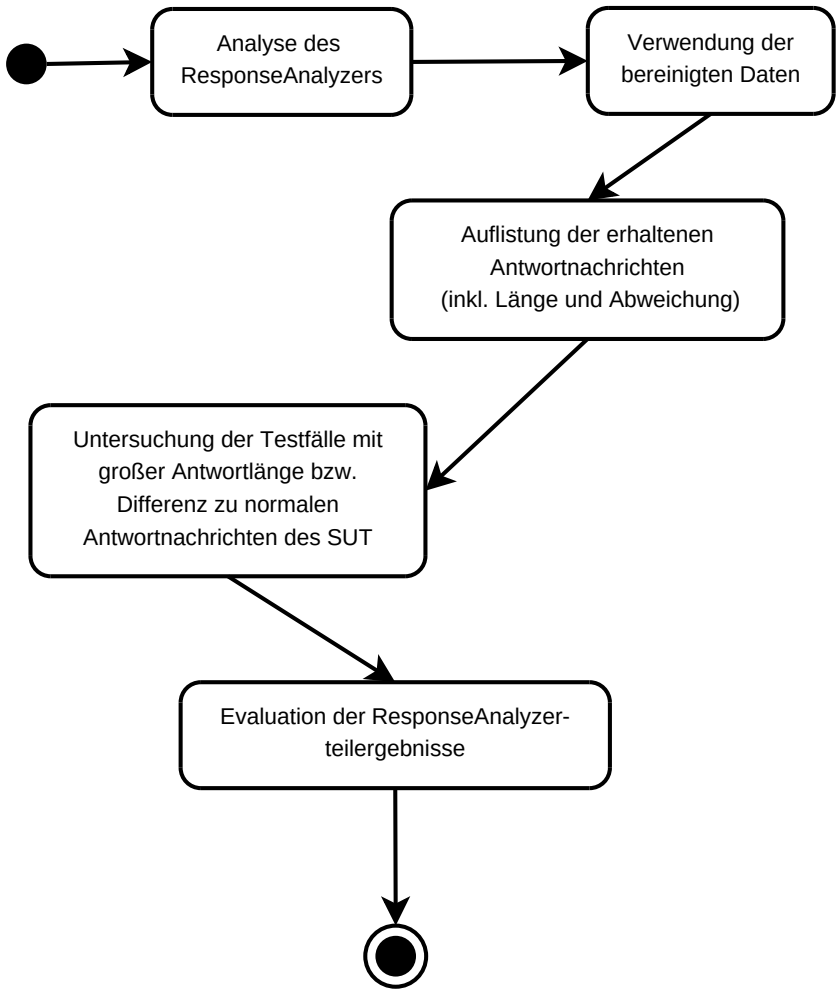

Abbildung 7.3: Schematische Darstellung des Analyseprozesses von Ergebnissen des ResponseAnalyzers

Das Abfrageergebnis zur Testfallbewertung durch den ResponseAnalyzer wird in Tabelle 7.7 schematisch dargestellt.

Testfall	Attacke	Typ	Länge	Abweichung
1	or	SQLInjection	523	12
2	%	CharacterInjection	3330	2229
3	034	RandomNumber	344	0
4	;bash &	CommandInjection	127	38
5	=!	SQLInjection	233	1887

Tabelle 7.7: Schematische Ermittlung der Länge pro erhaltener Antwortnachricht beim ResponseAnalyzer zur Überprüfung von Auffälligkeiten

Damit konnten jene Testfälle gefunden werden, bei welchen Abweichungen bezüglich der Antwortnachrichtlänge im Vergleich zu Testfällen zum normalen Verhalten des SUT aufgetreten sind. Es wurde ermittelt, ob der Fuzzer angenommen hatte, dass es sich um Sicherheitsfehler handelte. Hierbei konnte verifiziert werden, dass bei den vermuteten Sicherheitsfehlern nur falsch positive Ergebnisse vorlagen.
Der gleiche Schritt wurde mit dem ResponseTimeAnalyzer durchgeführt, wobei jener den Fokus auf die Antwortzeit der zu testenden Applikation bei seinen Bewertungen legt. Die schematische Vorgehensweise wird durch Abbildung 7.4 verdeutlicht.

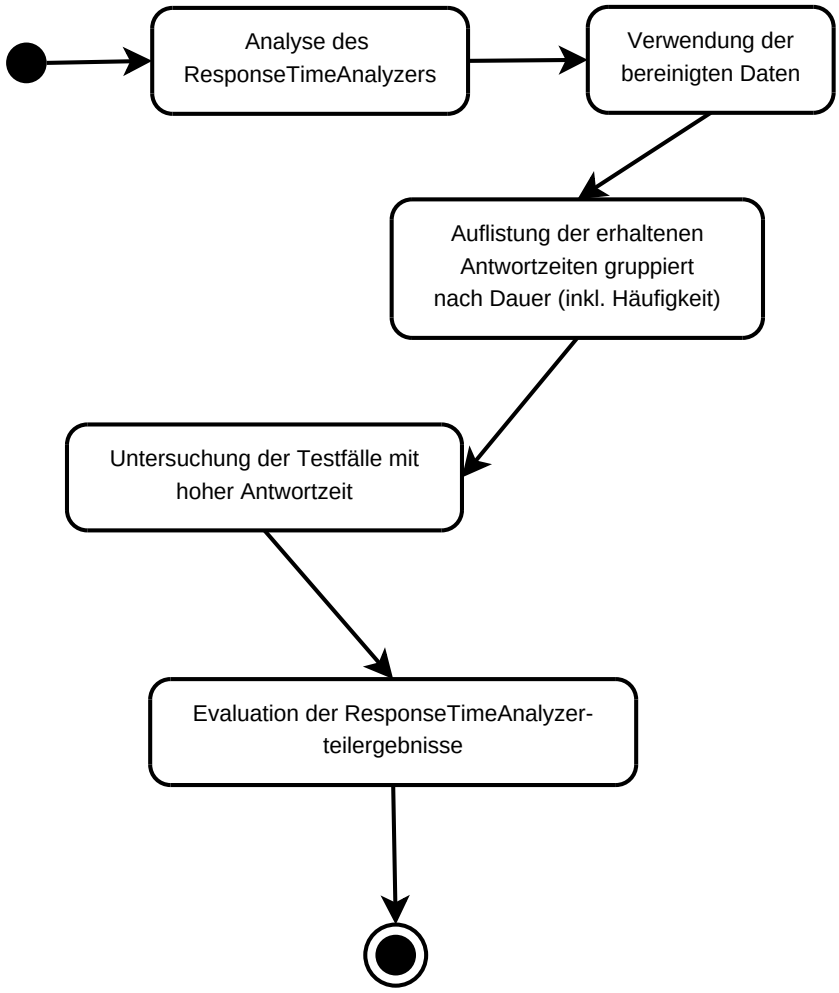

Abbildung 7.4: Schematische Darstellung des Analyseprozesses von Ergebnissen des ResponseTimeAnalyzers

Es wurden jene Testfallergebnisse mit einer hohen Antwortzeit aus den Daten extrahiert. Außerdem wurde untersucht, ob diese Testfälle möglicherweise Sicherheitsfehler bei der zu testenden Applikation erzeugt haben, das wird exemplarisch in der Tabelle 7.8 veranschaulicht. Dazu wurden die Testfälle in der Tabelle nach deren Antwortzeit gruppiert.

Gruppe	Antwortzeit	Häufigkeit
1	3043	4
2	3034	1
3	3033	2
4	3030	1
5	817	3
6	445	52
7	285	23
8	141	7

Tabelle 7.8: Schematische Ermittlung der Antwortzeit gruppiert nach der Häufigkeit beim ResponseTimeAnalyzer zur Überprüfung von Auffälligkeiten

Anschließend wurde überprüft, ob der Fuzzer die Nachrichten mit hoher Antwortzeit auch als potenzielle Sicherheitsfehler eingestuft hat, wie das Tabelle 7.9 schematisch zeigt.

Testfall	Attacke	Typ	Dauer	Abweichung
1	&#60	Characters	3043	2743
2	´	Characters	3043	2749
3	!#0%#̂0##0	CommandInjection	3033	532
4	#xA	Characters	3030	2758
5	..%bg%qf	Characters	817	2747
6	&#x03C	Charcters	285	1228
7	<	SQLInjection	445	26

Tabelle 7.9: Schematische Ermittlung der Dauer pro erhaltener Antwortnachricht beim ResponseTimeAnalyzer zur Überprüfung von Auffälligkeiten

Auch hier konnte festgestellt werden, dass die vom Fuzzer als potenzielle Sicherheitsfehler kategorisierten Testfallergebnisse falsch positiv waren.
Die schematische Darstellung der Analyse des ResponseClassificationAnalyzers wird nicht weiter verfolgt, da sie analog zu den anderen Analyzern stattgefunden hat.

Bei getrennter Betrachtung der einzelnen Analyzerergebnisse ist die Problematik aufgetreten, dass die untersuchten Analyzer (FileReader-, ResponseTime- und ResponseAnalyzer) teilweise unterschiedliche Ergebnisbewertungen geliefert haben. Zum Beispiel hat der FileReaderAnalyzer bei einem konkreten Testfall keine neuen, auffälligen Fehlermeldungen entdeckt und diesen somit nicht als potenziellen Sicherheitsfehler eingestuft. Der ResponseTimeAnalyzer hat bei demselben Testfall jedoch festgestellt, dass sich die Länge der Antwortzeit der zu testenden Applikation außerhalb eines bestimmten Intervalls befunden hatte. Dieser Testfall wurde von ihm somit als möglicher Sicherheitsfehler eingestuft. Das bedeutet, dass in diesem Fall zwei Analyzer zum exakt selben Testfall unterschiedliche Bewertungen abgegeben haben und der Fuzzer im Endeffekt entschieden hat, dass es sich dabei um einen Sicherheitsfehler gehandelt hat. Dies ist daher auf die Gewichtung der einzelnen Analyzer zurückzuführen. Der TcpPortAnalyzer war

bei den durchgeführten Auswertungen unauffällig und lieferte immer das Ergebnis, dass der Port nach jedem Testfall verfügbar war. Die Aussage dieses Analyzers konnte daher vernachlässigt werden. Memory- und CpuLoadAnalyzer konnten ebenso vernachlässigt werden, da der Fuzzer und das zu testende System nicht auf derselben physischen Maschine installiert gewesen und somit keine Ergebnisse diesbezüglich vorgelegen sind.
Eine gesonderte Betrachtung der Analyzer hat die Evaluation ziemlich erschwert, daher war eine ganzheitliche, analyzerübergreifende Analyse anzustreben. Diese wird im nächsten Abschnitt genauer beschrieben.

Klassifikation

Wie bereits im letzten Abschnitt erwähnt wurde, kann auf Grund der Betrachtung der einzelnen Analyzer nur schwer festgestellt werden, ob der Fuzzer die Bewertung seiner Analyzer korrekt verarbeitet hat. Es ist somit notwendig die Daten in ihrer Gesamtheit zu überprüfen. Der Ablauf dieses Vorgangs wird exemplarisch mit Hilfe der Abbildung 7.5 gezeigt.

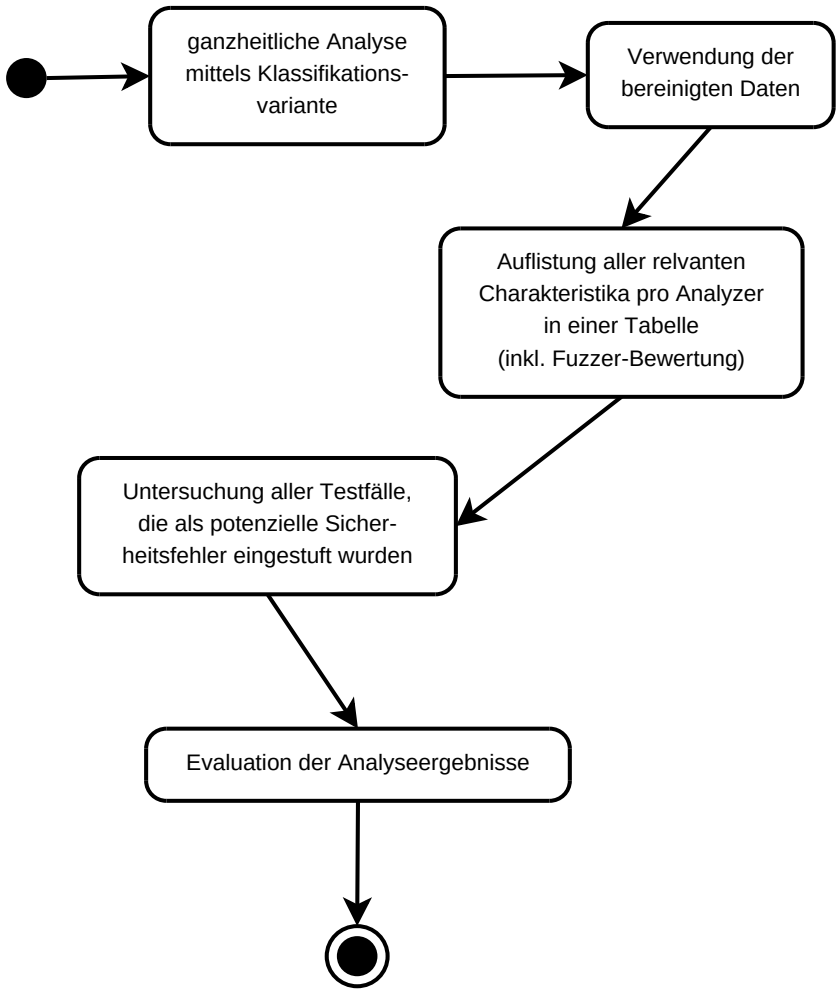

Abbildung 7.5: Schematische Darstellung des ganzheitlichen Analyseprozesses der einzelnen Analyzerergebnisse

Dazu wurde der Ansatz einer nachträglichen Klassifikation gewählt. Die Klassifikation mittels Entscheidungsbaum aus Abschnitt 6.2.2 und der Ansatz nach der Bayes aus Abschnitt 6.2.3 konnten nicht eins zu eins übernommen werden, um Sicherheitsfehler in den Daten zu finden. Diese Feststellung kommt daher, weil die Testfallergebnisse bereits vom Fuzzer vorklassifiziert (vorläufige Bewertung von möglichen Sicherheitsfehlern) wurden. Eine Klassifikation mittels

Algorithmus ist deshalb nicht sinnvoll, weil nur die vom Fuzzer bewerteten Ergebnisse vorliegen, aber eigentlich nie gesagt werden kann, ob manche Testfälle am SUT tatsächlich Sicherheitsfehler ausgelöst haben, da auf das SUT (im Normalfall) nicht zugegriffen werden kann. Der Klassifikationsalgorithmus benötigt aber ein richtig klassifiziertes Lernschema, mit welchem er die Klassifikation beginnen kann. [vgl. Abschnitt 6.2.2] Es ist somit nur eine Kontrolle der klassifizierten Testfälle möglich. Daher wurde eine modifizierte Variante der Klassifikation verwendet, welche sämtliche durch den Fuzzer eingestuften Sicherheitsfehler findet und verifiziert, ob der Fuzzer richtig klassifiziert hat. Zur Überprüfung der Richtigkeit der Fuzzerbewertung wurde folgende Abfrage erstellt, welche das Listing 7.1 zeigt.

```
1 SELECT t1.tc_id, t1.test_type_name, t1.attack, t3.ANALYZER_VALUE, ↩
    t3.ANALYZER_VALUE_DIFFERENCE_TO_LEARNING_PHASE,
2 t2.DURATION, t2.duration_difference, t4.LENGTH, t4.↩
    length_difference, t1.FINAL_RESULT
3 FROM ETL_RESPONSETIMEANALYZER t2
4 JOIN ETL_TESTCASE_VARIATION t1 ON t2.tc_id = t1.tc_id JOIN
5 ETL_ANALYZER_RESULT t3 ON t1.tc_id = t3.tc_id
6 JOIN ETL_RESPONSEANALYZER t4 ON t1.tc_id = t4.tc_id
7 where t3.ANALYZER_NAME = 'FileReaderAnalyzer' and t1.is_valid = ↩
    false and t1.FINAL_RESULT > 0
8 ORDER BY 10 desc;
```

Listing 7.1: Schematische Darstellung einer Abfrage zur Überprüfung der gesamten durch den Fuzzer erstellten Fehlerklassifikation

Zur besseren Veranschaulichung folgt eine exemplarische Darstellung der in Listing 7.1 beschriebenen Abfrage in Tabelle 7.10. Hierbei stehen die Abkürzung FRA, RA und RTA in der Tabelle jeweils für die einzelnen Analyzer, nämlich: FileReaderAnalyzer (FRA), ResponseAnalyzer (RA) sowie ResponseTimeAnalyzer (RTA). Die Spalte "Resultat" enthält jeweils immer die Fuzzergesamtbewertung pro Testfallergebnis.

Testfall	Attacke	FRA	RA	RTA	Resultat
1	&#60	apache; exception; userexception	817	23	B
2	..%bg%qf	java.lang; numberformatexception	3034	2749	A
3	!#xA	apache; exception	200	1	D
4	<	exception; userexception	3030	2758	A
5	<blank>	numberformatexception; saxparseexception	122	23	D
6	"	userexception	285	1228	C
7	4&	numberformatexception; userexception	445	35	B

Tabelle 7.10: Tabelle zur Überprüfung der gesamten Fehlerklassifikation durch den Fuzzer

Es wurde pro Testfall versucht auf Grund von Kriterien wie Antwortzeiten, Antwortnachrichtenlängen und gefundenen Fehlermeldungen herauszufinden, als wie wahrscheinlich der Fuzzer jene Testergebnisse als Sicherheitsfehler eingestuft hat. Bei der genauen Überprüfung der Ergebnisse konnte festgestellt werden, dass sie der Fuzzer als falsch positiv klassifiziert hat. Es ist an dieser Stelle anzumerken, dass mit der Abfrage aus Listing 7.1 die gleichen Testfälle gefunden werden konnten, wie bei der Analyse der einzelnen Analyzer. [vgl. Abschnitt 7.3] Die Evaluation dieses ganzheitlichen Ergebnisses wird gesondert im nächsten Abschnitt diskutiert.

7.4 Evaluation der Analyseergebnisse

Aus dem Analyzeprozess des Datenbestandes konnte folgendes ermittelt werden: Einerseits wurde die Analyzerbewertung der Testfälle einzeln betrachet und andererseits die Bewertung des Fuzzers in Summe. Dies ist mit Hilfe der in Abschnitt 7.3 vorgestellten Vorgangsweise geschehen.
Für den Proof-Of-Concept wurden kleine Ergebnismengen verwendet, um die Überprüfbarkeit zu gewährleisten. Für weitere Analysen in zukünftigen Arbeiten sollten größere Testergebnisse verwendet werden. Es wurden die Testergebnisse von drei Testläufen mit rund 8000 Testfällen analysiert. Sie werden mit ihrer Testanzahl und den falsch positiven Ergebnissen schematisch in Tabelle 7.11 dargestellt.

Testlauf	Testfallanzahl	falsch positive Ergebnisse
1	1509	197 (ca. 8%)
2	193	25 (ca. 13%)
3	5681	2184 (ca. 39%)

Tabelle 7.11: Schematische Tabelle mit den analysierten Testläufen sowie deren Testfallanzahl und falsch positiven Ergebnissen

Die Tabelle 7.12 zeigt die Aufschlüsselung der falsch positiven Ergebnisse auf die einzelnen Analyzer pro Testlauf. Dabei stehen FRA, RA, RCA, RTA jeweils für FileReaderAnalyzer, ResponseAnalyzer, ResponseClassificationAnalyzer, ResponseTimeAnalyzer.

Testlauf	falsch positive Ergebnisse	FRA	RA	RCA	RTA
1	197	190	0	0	8
2	25	24	0	0	1
3	2184	58	0	2153	0

Tabelle 7.12: Schematische Tabelle mit der Aufschlüsselung der falsch positiven Ergebnisse pro Testlauf und Analyzer

Beim ersten Testlauf sind 197 falsch positive Testfälle aufgetreten, wobei 190 davon vom FileReaderAnalyzer gefunden wurden. Dies ist darauf zurückzuführen, dass einige Fehlermeldungen gefunden werden konnten. ResponseAnalyzer und ResponseClassificationAnalyzer haben keine Auffälligkeiten entdeckt. Der ResponseTimeAnalyzer hat bei acht Testfällen eine erhöhte Antwortzeit festgestellt. Dies ist vermutlich dadurch bedingt, dass andere Prozesse am SUT auch abgearbeitet werden mussten und daher die Antwort nicht rechtzeitig an den Fuzzer geschickt wurde. Bei einem Testfall gab es eine Überschneidung zwischen ResponseTimeAnalyzer und FileReaderAnalyzer. Dieser Testfall wurde gesondert überprüft und konnte ebenso als falsch positives Ergebnis evaluiert werden.
Beim zweiten Testlauf ergaben sich 25 falsch positive Testfälle. Hierbei verhielten sich der ResponseAnalyzer und ResponseClassificationAnalyzer unauffällig. Der FileReaderAnalyzer sowie der ResponseTimeAnalyzer haben Sicherheitsfehler vermutet. Es gab bei diesem Testlauf zwischen den Analyzern im Vergleich zum ersten Testlauf keine Überschneidungen.
Beim dritten Testlauf hingegen war sehr auffällig, dass der ResponseClassificationAnalyzer 2153 Testfallergebnisse als Sicherheitsfehler eingestuft hat. Der FileReaderAnalyzer hat 58 Testfälle als Sicherheitsfehler klassifiziert. Zwischen den beiden Analyzern gab es bei insgesamt 27 Testfällen Überschneidungen. Die anderen Analyzer haben sich bei diesem Testlauf unauffällig verhalten.
Die Auffälligkeit, das beim dritten Testlauf der ResponseClassificationAnalyzer deratig viele

falsch positive Ergebnisse generiert hat, kann vermutlich dadurch erklärt werden, dass dieser Analyzer entweder zu sensibel gegenüber Schwankungen (in Bezug auf die erhaltene Antwort vom SUT) konfiguriert worden ist, dass die zu testende Applikation in der Testphase auf Grund von Angriffen anders reagiert (im Gegensatz zum normalen Verhalten) oder, dass die Gewichtung dieses Analyzers in Vergleich zu den anderen zu hoch eingestellt war. Dadurch ist ersichtlich, dass mit Hilfe der angewendeten Analyse auch Aussagen bezogen auf die konkrete Fuzzerkonfiguration gegeben werden. Zum gesamten Testergebnis kann gesagt werden, dass der Fuzzer in zwischen 8 und 40% der Testfälle falsch positiv bewertet hat.
Generell ist zu sagen, dass der Fuzzer immer einen Sicherheitsfehler vermutet hat, falls bei den Testfallergebnissen Unterschiede zu gängigen Antwortnachrichten des SUT bestanden haben, unabhängig vom Analyzer. Bei der Ergebnisbewertung des Fuzzers hat dieser Umstand sowie möglicherweise die Konfiguration der einzelnen Analyzer (wie stark diese auf Schwankungen reagieren) und deren Gewichtung letztendlich zu den falsch positiven Ergebnissen geführt. Es wird daher empfohlen mehrere Testläufe mit dem gleichen SUT und einer nachgelagerten Analyse durchzuführen. Diese Testläufe sollten jeweils mit einer unterschiedlichen Konfiguration der einzelnen Analyzer erfolgen. Es könnten infolgedessen möglicherweise konkrete Aussagen getroffen werden, ob einzelne Analyzer vor den Testläufen misskonfiguriert worden sind oder nicht. Außerdem könnte eventuell genauer festgestellt werden, ob es sich nur um zufällige Ausreißer bei Analyzerbewertungen (z. B. der ResponseClassificationAnalyzer) wie in Testlauf drei handelt oder ob der Analyzer richtig eingestellt war und die Applikation möglicherweise tatsächlich Sicherheitsfehler enhält, die bei einem einmaligen Testlauf als solche nicht entdeckt worden wären.
Im Rahmen dieser Arbeit konnten keine Abhängigkeiten zwischen Testfallergebnissen festgestellt werden, obwohl die Ergebnisse auf charakteristische Eigenschaften hin untersucht worden sind, wie zum Beispiel speziell die Überschneidung der Analyzerbewertungen aus Testlauf eins und drei. Die Testfälle konnten im Zuge der Analyse auch nicht in bestimmte Kategorien eingeteilt werden, außer positiv und falsch positiv. Die Voraussetzung für ein Vorhaben zur Kategorisierung von Testfallergebnissen wäre ein besseres Datenmodell, welches die Daten nach anderen Gesichtspunkten aufgeschlüsselt enthält, wie zum Beispiel eine Gruppierung von Testfällen mit ähnlichen Attack-Vektoren. Die Kritikalität von potenziellen Sicherheitsfehlern wurde vom Fuzzer bereits vor der Analyse bestimmt, pro Testfall mittels Gewichtung der Analyzer und den Abweichungen zum normalen Verhalten des SUT (=Fuzzer-Bewertung). Eine nachträgliche Klassifikation von Sicherheitsfehlern bezüglich deren Kritikalität wäre daher beim vorliegenden Testergebnis nicht zielführend gewesen, da nicht gesagt werden kann, ob ein vom Fuzzer vermuteter Sicherheitsfehler mit der Bewertung 0.7 in Wirklichkeit einen kritischeren Fehler dargestellt hat als ein potenzieller Sicherheitsfehler, der vom Fuzzer mit 0.4 bewertet worden ist. Zur Häufigkeit der aufgetretenen falsch positiven Sicherheitsfehler ist zu sagen, dass diese vermutlich stark von der Konfiguration und Gewichtung der einzelnen Analyzer abhängt sowie auch von der Anzahl der durgeführten Testläufe respektive Testfälle pro Testlauf. Es kann daher keine genaue Häufigkeitsangabe über potenzielle Sicherheitsfehler gemacht werden.

8 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es Sicherheitstestergebnisse mittels Data-Mining-Methoden zu analysieren und zu optimieren. Als Einführung wurden die Themenbereiche Softwaresicherheit im Softwareentwicklungsprozess und mögliche Sicherheitsbedrohungen im Softwareumfeld erklärt.
Im Zuge von Softwaretests wurde näher auf die verschiedenen Aspekte beim Testen eingegangen. Ebenso wurde ein Prozess vorgestellt nach dem Software schematisch getestet werden kann. Verschiedene Teststufen wurden erläutert und die Wichtigkeit von Sicherheitstests, welche in Verbindung mit Fuzzing gebracht wurde. Dabei bietet Fuzzing als eine Testmethode verschiedene Möglichkeiten Software auf Sicherheitsfehler zu überprüfen.
Sicherheitstests liefern unterschiedliche Testergebnisse, welche analysiert werden können. Im Bereich Data-Mining gibt es Methoden, welche eine solche Analyse ermöglichen. Zuerst wurde allgemein eine Einführung in Data-Mining gegeben. Es wurde erläutert, wie Wissen vorbereitet, dargestellt und anschließend vearbeitet werden kann. Anschließend wurde das in dieser Arbeit verwendete Fuzzing-Framework vorgestellt. Es wurde dessen Architektur, Datenbankstruktur und die Zusammenhänge zu Sicherheitsbedrohungen erläutert. Als Nächstes wurde der Fokus auf den Bereich Data-Mining in Verbindung mit Softwaresicherheit gelenkt. Auf Grund verschiedener Literaturrecherchen wurden vier Data-Mining-Konzepte gewählt, welche schematisch an Beispielen erklärt wurden. Die vier Ansätze waren Association-Mining mittels Apriori-Algorithmus, Klassifikation mittels Entscheidungsbäumen und nach Bayes, Clusteranalyse mittels K-Means-Methode und Text-Mining.
Im Rahmen einer praktischen Analyse vorhandener Testergebnisse wurde als Proof-Of-Concept überprüft, welche Aussagen mittels Data-Mining-Techniken getroffen werden konnten. Zuerst wurde eine Analyse des Datenbestandes ausgehend von den einzelnen Analyzern des Fuzzers durchgeführt. Diese isolierte Betrachtung des Ergebnisses war jedoch nicht ausreichend. Eine gesamtheitliche Analyse wurde daher angestrebt und mit einer abgewandelten Form der Klassifikation durchgeführt. Dabei wurden Testfallergebnisse gefunden, welche der Fuzzer als potenzielle Sicherheitsfehler eingestuft hatte. Die Evaluation hat aber ergeben, dass es sich bei diesen Testfällen um falsch positive Bewertungsergebnisse des Fuzzers gehandelt hat.
Der Proof-Of-Concept hat ergeben, dass die vorhandene Modellierung der Daten für umfangreiche Analysen nicht ausreichend ist. Als weiterführende Arbeit sollte das Datenmodell des Fuzzing-Werkzeugs überarbeitet und auf entsprechende Data-Mining-Tätigkeiten erweitert werden. Für zukünftige Testläufe wird empfohlen, beim ersten Testlauf eine zufällige Gewichtung der Analyzer zu wählen. Danach kann eine Erstanalyse der Daten erfolgen. Nachdem die Ergebnisse der Analyse vorliegen, ist eine Neugewichtung der Analyzer vorzunehmen. Anschließend wird wieder ein Testlauf gestartet. Dieser Prozess kann beliebig oft wiederholt werden, um so herauszufinden, ob sich die Ergebnisaussage des Fuzzers von Testlauf zu Testlauf im Vergleich zu vorherigen Testläufen verbessert oder verschlechtert hat. Somit könnte auch festgestellt werden, ob sich die Anzahl der falsch positiven Testfallergebnisse möglicherweise zwischen den Testläufen verringert oder vermehrt hat.
Es kann für zukünftige Analyse- und Optimierungsprozesse auch noch der Ansatz mittels Text-Mining und Clusteranalyse erprobt werden. Hierbei können zum Beispiel alle Testfälle (unabhängig davon, ob Sicherheitsfehler oder nicht) in ähnliche Cluster eingeteilt werden, um so eine Aussage darüber zu treffen, ob sich hinter manchen Clustern potenzielle Sicherheitsfehler oder Systemabstürze verbergen.

Literatur

[1] B. B. Agarwal, Tayal S. P. und M. Gupta. *Software Engineering & Testing*. Jones & Bartlett Learning, 2011.

[2] A. Ahmed. *Software Testing as a Service*. Auerbach Publications, 2009.

[3] J. Allen u. a. *Software Security Engineering: A Guide for Project Managers*. Addison-Wesley Professional, 2008.

[4] P. Ammann und J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[5] T. Atanasova u. a. "Analysis of the possible application of Data Mining, Text Mining and Web Mining in Business Intelligent Systems". In: *2010 Proceedings of the 33rd International Convention MIPRO* (2010), S. 1294–1297.

[6] D. Baca und B. Carlsson. "Agile Development with Security Engineering Activities". In: *ICSSP '11 Proceedings of the 2011 International Conference on on Software and Systems Process* (2011), S. 149–158.

[7] S. Bekrar u. a. "Finding Software Vulnerabilities by Smart Fuzzing". In: *ICST'11*. 2011, S. 427–430.

[8] M. Berry und G. Linoff. *Data Mining Techniques*. 2nd. Wiley Publishing, Inc., 2004.

[9] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2002.

[10] M. Bramer. *Principles of Data Mining*. Springer, 2007.

[11] D. Byers und N. Shahmehri. "Prioritisation and Selection of Software Security Activities". In: *ARES '09. International Conference on Availability, Reliability and Security, 2009.* (2009), S. 201–207.

[12] W. Cheng-Chih, C. Kuan-Chou und H. Hui-Min. "Associational Approach of Text Data Mining and Its Implications". In: *2004 IEEE International Conference on Networking, Sensing and Control* Vol. 1 (2004), S. 243–248.

[13] B. Chess und B. Arkin. "Software Security in Practice". In: *Security & Privacy, IEEE* 9 Issue 2 (2011), S. 89–92.

[14] B. Chess und J. West. *Security Programming with Static Analysis*. Addison-Wesley Professional, 2007.

[15] *Common Criteria for Information Technology Security Evaluation, version 3.1, revision 3 final, 2009*.

[16] H. Dai, C. Murphy und G. Kaiser. "Configuration Fuzzing for Software Vulnerability Detection". In: *ARES '10 International Conference on Availability, Reliability, and Security, 2010* 525-530 (2010).

[17] T. Dasu und T. Johnson. *Exploratory Data Mining and Data Cleaning*. Wiley-Interscience, 2003.

[18] G. Di Fatta, S. Leue und E. Stegantova. "Discriminative Pattern Mining in Software Fault Detection". In: *SOQUA '06 Proceedings of the 3rd international workshop on Software quality assurance* (2006), S. 62–69.

[19] H. A. Do Prado und E. Ferneda. *Emerging Technologies of Text Mining: Techniques and Applications*. Idea Group Reference, 2007.

[20] M. Dowd, J. McDonald und J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006.

[21] J. Du, Y. Yang und Q. Wang. "An Analysis for Understanding Software Security Requirement Methodologies". In: *SSIRI 2009. Third IEEE International Conference on Secure Software Integration and Reliability Improvement, 2009.* (2009), S. 141–149.

[22] J. Erickson. *Hacking - The Art of Exploitation*. No Starch Press, 2007.

[23] A. Esfandi. "Efficient Anomaly Intrusion Detection System in Adhoc Networks by Mobile Agents". In: *2010 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT)* (2010), S. 73–77.

[24] G. D. Everett und R. McLeod Jr. *Software Testing: Testing Across the Entire Software Development Life Cycle*. Wiley-IEEE Computer Society Pr, 2007.

[25] P. Farrell-Vinay. *Manage Software Testing*. Auerbach Publications, 2008.

[26] D. G. Firesmith. *Common Concepts Underlying Safety, Security, and Survivability Engineering*. Techn. Ber. Carnegie Mellon Software Engineering Institute, 2003.

[27] R. Gabriel, T. Hoppe und A. Pastwa. "Analyzing Malware Log Data to Support Security Information and Event Management: Some Research Results". In: *DBKDA '09. First International Conference on Advances in Databases, Knowledge, and Data Applications* (2009), S. 108–113.

[28] Z. Gang u. a. "An Heuristic Method for Web-Service Program Security Testing". In: *CHINAGRID '09 Proceedings of the 2009 Fourth ChinaGrid Annual Conference* (2009), S. 139–144.

[29] P. Giudici. *Applied Data Mining Statistical Methods for Business and Industry*. Wiley, 2003.

[30] D. Graham u. a. *Foundations of Software Testing*. Cengage Learning Business Press, 2006.

[31] F. Guillet und H. Hamilton. *Quality Measures in Data Mining*. Springer, 2007.

[32] G. K. Gupta. *Introduction to Data Mining with Case Studies*. Eastern Economy. Prentice-Hall of India Pvt.Ltd, 2006.

[33] V. Gupta und G. S. Lehal. "A Survey of Text Mining Techniques and its Applications". In: *JOURNAL OF EMERGING TECHNOLOGIES IN WEB INTELLIGENCE* 1 (2009).

[34] S. T. Halkidis u. a. "Architectural Risk Analysis of Software Systems Based on Security Patterns". In: *IEEE Transactions on Dependable and Secure Computing* 5 Issue 3 (2008), S. 129–142.

[35] J. Han und M. Kamber. *Data Mining Concepts and Techniques*. 2nd. Morgan Kaufmann Publishers, 2006.

[36] J. Han u. a. "Frequent pattern mining: current status and future directions". In: *Springer Science+Business Media* 15 Issue 1 (2007), S. 55–86.

[37] D. Hand, H. Mannila und P. Smyth. *Principles of Data Mining*. The MIT Press, 2001.

[38] A. M. J. Hass. *Guide to Advanced Software Testing*. Artech House Publishers, 2008.

[39] P. Herzog. *OSSTMM - Open Source Security Testing Methodology Manual. Last available in November 2011 from www.isecom.org/osstmm/ and http://security4internet.net/*.

[40] R. Hewett. "Mining software defect data to support software testing management". In: *Kluwer Academic Publishers Hingham* 34 Issue 2 (2011), S. 245–257.

[41] R. Hewett u. a. "Software Defect Data and Predictability for Testing Schedules". In: *SEKE*. 2006.

[42] M. Howard, D. LeBlanc und J. Viega. *24 Deadly Sins of Software Security*. McGraw-Hill Osborne Media, 2009.

[43] Z. Hui u. a. "A Taxonomy of Software Security Defects for SST". In: *2010 International Conference on Intelligent Computing and Integrated Systems (ICISS)* (2010), S. 99–103.

[44] M. Kantardzic. *Data Mining: Concepts, Models, Methods and Algorithms*. Wiley-IEEE Press, 2002.

[45] P. Khadivi und M. Momtazpour. "Cipher-Text Classification with Data Mining". In: *2010 IEEE 4th International Symposium on Advanced Networks and Telecommunication Systems (ANTS)* (2010), S. 64–66.

[46] W. E. Lewis. *Software Testing and Continuous Quality Improvement*. 3rd. Auerbach Publications, 2008.

[47] H. Li u. a. "The Merging Trend of Software Security and Safety". In: *2011 International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (ICQR2MSE)* (2011), S. 219–222.

[48] L. Li und D. Xiao. "Research on The Network Security Management Based on Data Mining". In: *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)* 5 (2010), S. 184–187.

[49] N. Li, Z. Li und X. Sun. "Classification of Software Defect Detected by Black-box Testing: An Empirical Study". In: *World Congress on Software Engineering* 2 (2010), S. 234–240.

[50] N. Li, Z. Li und L. Zhang. "Mining Frequent Patterns from Software Defect Repositories for Black-box Testing". In: *2nd International Workshop on Systems and Applications (ISA)* 1 (2010).

[51] M. A. van der Linden. *Testing Code Security*. Auerbach Publications, 2007.

[52] G. Lv u. a. "Information Security Monitoring System based on Data Mining". In: *2009 Fifth International Conference on Information Assurance and Security* (2009), S. 472–475.

[53] C. Manning, P. Raghavan und H. Schuetze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

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

[55] G. J. Myers. *The Art of Software Testing*. Hrsg. von Corey Sandler, Tom Badgett und Todd M. Thomas. Wiley, 2nd Edition, 2004.

[56] N. K. Nagwani und A. Bhansali. "A Data Mining Model to Predict Software Bug Complexity Using Bug Estimation and Clustering". In: *Proceedings of the 2010 International Conference on Recent Trends in Information, Telecommunication and Computing* (2010), S. 13–17.

[57] Y. Nong. *The Handbook of Data Mining*. CRC Press, 2003.

[58] F. J. B. Nunes, A. D. Belchior und A. B. Albuquerque. "Security Engineering Approach to Support Software Security". In: *2010 6th World Congress on Services (SERVICES-1)* (2010), S. 48–55.

[59] P. Oehlert. "Violating Assumptions with Fuzzing". In: *Security & Privacy, IEEE* (2005), S. 58–62.

[60] O. Parr-Rud. *Data Mining Cookbook*. Wiley, 2000.

[61] M. Pezze und M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[62] M. F. Porter. “An algorithm for suffix stripping”. In: *Programm 14* 3 (1980), S. 130–137.

[63] M. F. Porter. *Porter Stemmer in Java. Last available in November 2011 on*. URL: http://tartarus.org/~martin/PorterStemmer/.

[64] S. Qinbao u. a. “Software Defect Association Mining and Defect Correction Effort Prediction”. In: *IEEE Transactions on Software Engineering* 32 Issue 2 (2006), S. 69–82.

[65] A. Rajaraman und J. Ullman. *Mining of Massive Datasets*. Ullman, 2010.

[66] D. Sanchez, M. J. Martin-Bautista und Blanco I. “Text Knowledge Mining: An Alternative to Text Data Mining”. In: *ICDMW ’08 Proceedings of the 2008 IEEE International Conference on Data Mining Workshops* (2008), S. 664–672.

[67] C. Schanes u. a. “Security Test Approach for Automated Detection of Vulnerabilities of SIP-based VoIP Softphones”. In: *International Journal On Advances in Security* 4, no 1 & 2 (2011), S. 95–105.

[68] S. Sumathi und S. N. Sivanandam. *Introduction to Data Mining and its Applications*. Springer, 2006.

[69] M. Sutton, A. Greene und P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[70] S. Taber u. a. “Automated Security Test Approach for SIP-based VoIP Softphones”. In: *2010 Second International Conference on Advances in System Testing and Validation Lifecycle (VALID)* (2010), S. 114–119.

[71] A. Takanen, J. DeMott und C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House Publishers, 2008.

[72] B. Thuraisingham u. a. “Data Mining for Security Applications”. In: *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing* 3917/2006 (2008), S. 585–589.

[73] L. Wang und F. Xiuju. *Data Mining with Computational Intelligence*. Springer, 2005.

[74] J. Whittaker. *Exploratory Software Testing*. Addison-Wesley Professional, 2009.

[75] I. Witten, E. Frank und M. Hall. *Data Mining Pracital Machine Learning Tools and Techniques*. 3rd. Morgan Kaufmann Publishers, 2010.

[76] K. R. van Wyk und G. McGraw. “Bridging the Gap between Software Development and Information Security”. In: *Security & Privacy, IEEE* 3 Issue 5 (2005), S. 75–79.

[77] A.-U.-H. Yasar u. a. “Best Practices for Software Security: An Overview”. In: *INMIC 2008. IEEE International Multitopic Conference, 2008* (2008), S. 169–173.